

(12) INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(19) World Intellectual Property Organization  
International Bureau



(43) International Publication Date  
20 September 2001 (20.09.2001)

PCT

(10) International Publication Number  
**WO 01/69887 A2**

(51) International Patent Classification<sup>7</sup>: **H04L 29/00**

(21) International Application Number: **PCT/US01/08533**

(22) International Filing Date: **16 March 2001 (16.03.2001)**

(25) Filing Language: **English**

(26) Publication Language: **English**

(30) Priority Data:  
60/190,358 16 March 2000 (16.03.2000) US  
09/728,253 1 December 2000 (01.12.2000) US

(71) Applicant: **SRI INTERNATIONAL [US/US]; 333 Ravenswood Avenue, Menlo Park, CA 94025-3493 (US).**

(72) Inventor: **TEMPLIN, Fred, Lampert; 291 La Cuesta Drive, Portola Valley, CA 94028 (US).**

(74) Agent: **RODRIGUEZ, Michael, A.; Testa, Hurwitz & Thibault, LLP, High Street Tower, 125 High Street, Boston, MA 02110 (DE).**

(81) Designated States (*national*): AE, AG, AL, AM, AT, AU, AZ, BA, BB, BG, BR, BY, BZ, CA, CH, CN, CO, CR, CU, CZ, DE, DK, DM, DZ, EE, ES, FI, GB, GD, GE, GH, GM, HR, HU, ID, IL, IN, IS, JP, KE, KG, KP, KR, KZ, LC, LK, LR, LS, LT, LU, LV, MA, MD, MG, MK, MN, MW, MX, MZ, NO, NZ, PL, PT, RO, RU, SD, SE, SG, SI, SK, SL, TJ, TM, TR, TT, TZ, UA, UG, UZ, VN, YU, ZA, ZW.

(84) Designated States (*regional*): ARIPO patent (GH, GM, KE, LS, MW, MZ, SD, SL, SZ, TZ, UG, ZW), Eurasian patent (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), European patent (AT, BE, CH, CY, DE, DK, ES, FI, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE, TR), OAPI patent (BF, BJ, CF, CG, CI, CM, GA, GN, GW, ML, MR, NE, SN, TD, TG).

**Published:**

— *without international search report and to be republished upon receipt of that report*

*For two-letter codes and other abbreviations, refer to the "Guidance Notes on Codes and Abbreviations" appearing at the beginning of each regular issue of the PCT Gazette.*

WO 01/69887 A2

(54) Title: **AN IPv6-IPv4 COMPATIBILITY AGGREGATABLE GLOBAL UNICAST ADDRESS FORMAT FOR INCREMENTAL DEPLOYMENT OF IPv6 NODES WITHIN IPv4 NETWORKS**

(57) Abstract: Described is an IPv6-IPv4 compatibility global aggregatable unicast address format that enables incremental deployment of IPv6 hosts and routers within predominantly IPv4 based networks. A globally aggregatable IPv6 address prefix is associated with an IPv4 node having an IPv4 address and deployed in the network. The IPv4 node is configured with an IPv6-IPv4 compatibility address that includes a prefix portion and an interface identifier portion. The prefix portion contains the IPv6 address prefix associated with the IPv4 node and the interface identifier portion contains the IPv4 address. Packets addressed to the IPv6-IPv4 compatibility address of the IPv4 node are routed across IPv6 infrastructure using the IPv6 prefix portion or tunneled across IPv4 infrastructure using the IPv4 address embedded in the interface identifier portion.

**AN IPv6-IPv4 COMPATIBILITY AGGREGATABLE GLOBAL UNICAST ADDRESS  
FORMAT FOR INCREMENTAL DEPLOYMENT OF IPv6 NODES WITHIN IPv4  
NETWORKS**

**Related Application**

This application claims the benefit of the filing date of co-pending U.S. Provisional Application, Serial No. 60/190,358, filed March 16, 2000, entitled "An IPv4 Compatibility Aggregatable Global Unicast Address Format," the entirety of which provisional application is incorporated by reference herein.

5

**Government Support**

This invention was funded with government support under Contract No. DAAB07-96-D-H002, awarded by the U.S. Army Communications and Electronics Command. The United States government has certain rights to this invention.

**Background**

10

The global Internet infrastructure is comprised of communications entities (known as hosts, gateways and routers) that exchange messages using the Internet Protocol (IP). The current Internet Protocol, also known as IP Version 4 (or IPv4), has several limitations, including a 32-bit address space which limits the number of unique IPv4 addresses to approximately 4 billion. In practice, the actual number of unique addresses available is considerably less than 4 billion due to other limitations of the IPv4 addressing architecture.

15

For these reasons, the Internet Engineering Task Force (IETF) has developed a second-generation Internet Protocol known as IP Version 6 (or IPv6). IPv6 includes a 128-bit address space, which expands the number of uniquely addressable communications entities by orders of magnitude over the existing IPv4 addressing architecture. IPv6 also includes numerous

- 2 -

additional architectural improvements over the existing IPv4 protocol, such as address auto-configuration, neighbor discovery, and router discovery. The IPv4 protocol, however, is so well understood and so deeply entrenched in the existing Internet infrastructure that the transition to IPv6 will require an extended period of time during which IPv6 will initially coexist with and  
5 then gradually begin to supplant the existing IPv4 protocol.

For these reasons, an interim heterogeneous IPv6/IPv4 infrastructure is anticipated in which existing IPv4 entities retain their IPv4 addresses, but also are assigned IPv6 addresses such that these entities can communicate via both the IPv4 and IPv6 protocols. Current efforts to support IPv4/IPv6 coexistence focus on inter-domain routing between IPv6 "islands" using the  
10 existing global IPv4 backbone as transit. But, these islands themselves may consist of complex heterogeneous IPv4/IPv6 networks (e.g., large academic or commercial campus "intranets") that require intra-domain IPv4 to IPv6 transition mechanisms and strategies as well. Other approaches require statically configured "tunnels" across the legacy IPv4 infrastructure. These approaches are expensive in terms of configuration and operational administrative resources, and  
15 do not adapt to network topology changes. Another approach uses "dynamic tunneling," but requires the use of IPv6 multicast address. Thus, there remains the need for a mechanism that accommodates the transition of networks from IPv4 to IPv6 without the aforementioned problems.

- 3 -

**Summary of the Invention**

One objective of the invention is to facilitate the transition from current Internet Protocol (IPv4) to next-generation Internet Protocol (IPv6). Another objective is to provide addressing extensions to the existing IPv6 addressing architecture, rather than reinvent the IPv6 addressing scheme. Still another objective is to provide an IPv6-IPv4 compatibility aggregatable global unicast address format that greatly reduces the complexity for migration from the existing IPv4 Internet to IPv6 and allow Internet Service Providers (ISPs), corporations, and end-users to preserve their investments in legacy networking equipment during the transition phase between IPv4 and IPv6. Other objectives are for the address format to require no administrative overhead, to adapt seamlessly to network topology changes, and to work across a legacy IPv4 routing domain in which no IPv6 routing (either multicast or unicast) exists.

In one aspect, the invention features a method for enabling incremental deployment of Internet Protocol version 6 (IPv6) in a network having Internet Protocol version 4 (IPv4) nodes. A globally aggregatable IPv6 address prefix is associated with an IPv4 node having an IPv4 address and deployed in the network. The IPv4 node is configured with an IPv6-IPv4 compatibility address that includes a prefix portion and an interface identifier portion. The prefix portion contains the IPv6 address prefix associated with the IPv4 node and the interface identifier portion contains the IPv4 address. Packets addressed to the IPv6-IPv4 compatibility address of the IPv4 node are routed across IPv6 infrastructure using the IPv6 prefix portion or tunneled across IPv4 infrastructure using the IPv4 address embedded in the interface identifier portion.

In another aspect, the invention features an address format for achieving address compatibility between a previous version and a new version of Internet Protocol (IP). The address format comprises a prefix address portion that contains a globally aggregatable IP



- 4 -

address prefix and an interface identifier portion that contains an embedded IP address assigned according to the previous IP version. The address embedded in the interface identifier portion can be globally non-unique. The interface identifier portion is concatenated to the prefix portion to form an address that is compatible with the new IP version and enables packets to be routed to addresses assigned according to the previous IP version. The address format can include a type field that indicates whether the interface identifier portion contains an embedded IP address assigned according to the previous IP version. The address format enables routing nodes to route packets using the address format through nodes communicating according to the new IP version or to tunnel such packets through one or more nodes operating according to the previous IP version.

### **Brief Description of the Drawings**

The invention is pointed out with particularity in the appended claims. The objectives advantages of the invention described above, as well as further objectives and advantages of the invention, may be better understood by reference to the following description taken in conjunction with the accompanying drawings, in which:

Fig. 1 is a block diagram of an embodiment of a mobile internetworking system including a plurality of subnets in communication with the Internet;

Fig. 2 is a block diagram of a portion of an embodiment of protocol stack that can be implemented by each of the routing nodes in each subnet to communicate in accordance with the principles of the invention;

Fig. 3 is a flow diagram illustrating an embodiment of a process by which each routing node selects a parent neighbor node and children neighbor node(s) for each potential source node in the subnet to define a minimum-hop-path tree for each potential source node along which routing nodes receive and forward link-state updates originating from that source node;

- 5 -

Fig. 4 is a diagram illustrating an embodiment of an exemplary minimum-hop-path tree for the nodes in the subnet of Fig. 1;

Fig. 5 is a block diagram illustrating the operation of a partial topology version of the TBRPF protocol;

5 Fig. 6 is a diagram of an embodiment of a format of a message header for an atomic TBRPF protocol message;

Fig. 7 is a diagram of an embodiment of a format of a compound TBRPF message;

Figs. 8A and 8B are diagrams of embodiments of a format of a NEW PARENT protocol message;

10 Fig. 9 is a diagram of an embodiment of a format for a CANCEL PARENT message;

Figs. 10A and 10B are diagrams of embodiments of exemplary formats for link-state update messages;

Fig. 11 is a diagram of an embodiment of an exemplary format of a RETRANSMISSION\_OF\_BROADCAST message;

15 Fig. 12 is a flow diagram of an embodiment of a process performed by the nodes of the subnet to achieve neighbor discovery;

Fig. 13 is a diagram of a packet format for the protocol messages used for neighbor discovery;

20 Figs. 14 are a flow diagram of another embodiment of a process for performing neighbor discovery;

Fig. 15A is a diagram of a format for an IPv6 address including a prefix and an interface identifier;

Fig. 15B is a diagram of an embodiment of the interface identifier including a 24-bit company identifier concatenated with a 40-bit extension identifier;

- 6 -

Fig. 15C is a diagram of an embodiment of the interface identifier including a 24-bit company identifier concatenated with the 40-bit extension identifier;

Fig. 15D is a diagram of an IP v6-IPv4 compatibility address;

Fig. 15E is a diagram of an embodiment of a message format for tunneling an IPv6-IPv4 compatibility address through IPv4 infrastructure;

Fig. 16 is a flow diagram of an embodiment of a process by which a router tests an IPv6-IPv4 compatibility address;

Fig. 17 is a flow diagram of an embodiment of a process by which a mobile node and a server exchange messages;

Figs. 18A and 18B are diagrams illustrating an example of the operation of a message queue.

### **Description of the Invention**

Fig. 1 shows an embodiment of an internetworking system 2 including communication sub-networks ("subnets") 10, 20 that are components of a worldwide network of networks 30 (i.e., the "Internet"). The Internet 30 includes communications entities, (e.g., hosts and routers), that exchange messages according to an Internet Protocol (IP) such as IPv4 (version 4) and IPv6 (version 6). On the Internet 30, entities implementing IPv6 may coexist with IPv4 entities. In general, the IPv4 and IPv6 versions are incompatible. The incompatibility is due, in part, to the difference in addressing format: the IPv4 specifies a 32-bit address format, whereas the IPv6 specifies a 128-bit address format.

A server 40 is connected to the Internet 30 by a wire-line or wireless connection. The server 40 can be internal or external to the subnet 10. For purposes such as hosting application programs, delivering information or Web pages, hosting databases, handling electronic mail ("e-mail"), or controlling access to other portions of the Internet 30, the server 40 is a computer

- 7 -

system that typically handles multiple connections to other entities (e.g., client systems) simultaneously. Although represented as a single server 40, other embodiments can have a group of interconnected servers. The data on the server 40 are replicated on one or more of these interconnected servers to provide redundancy in the event that a connection to the server 40 cannot be established.

Each subnet 10, 20 includes one or more networks that can include both local area network (LAN) and wide area network (WAN) components. Each subnet 10, 20 may be a freely accessible component of the public Internet 30, or a private Intranet. The subnet 10 includes IP hosts 12, routers 14, and a gateway 16 (collectively referred to as nodes 18). As used hereafter, a router 14 is any node 18 that forwards IP packets not explicitly addressed to itself, and an IP host 12 is any node 18 that is not a router 14. Examples of devices that can participate as a node 18 in the subnet 10 include laptop computers, desktop computers, wireless telephones, and personal digital assistants (PDAs), network computers, television sets with a service such as Web TV, client computer systems, server computer systems. The gateway 16 is a particular type of routing node 14 that connects the subnet 10 to the Internet 30. The subnet 20 is similarly configured with nodes 18' (i.e., hosts 12', routers 14', and gateways 16').

The subnet 10 can be associated with one organization or administrative domain, such as an Internet service provider (ISP), which associates each node 18 with an assigned IPv6 or IPv4 network address. Each IPv6 address is globally unique, whereas each IPv4 address is locally unique at least within the subnet 10, and may be globally unique. Presumably, the assigned IP address has some topological relevance to the "home" subnet 10 of the node 18 so that the nodes 18 of the subnet 10 can be collectively identified by a common address prefix for routing purposes (called address aggregation). In one embodiment, the gateway 16 is a dual-stack node; that is, the gateway 16 has two IP addresses, an IPv6 address and an IPv4 address, and can route packets to IPv4 and IPv6 nodes.

- 8 -

Although it is conceivable that all nodes 18 in subnet 10 are initially assigned network addresses that follow a common address convention and have a common network prefix, dynamic topology changes may result in nodes 18 leaving their home subnet 10 to join a “foreign” subnet (e.g., subnet 20) and new nodes joining the home subnet 10. Because the nodes 5 18 maintain the same IP address irrespective of whether the node 18 changes its location within the subnet 10 or moves to the foreign subnet 20, mobility may result in a heterogeneous conglomerate of IPv6 and IPv4 addresses, having various network prefixes, within the single subnet 10 unless some form of dynamic address assignment or other address-renumbering scheme is used. Further, the gradual transition from the use of IPv4 network addresses to IPv6 10 network addresses within the subnet 10 increases the likelihood of such a heterogeneous conglomeration. Thus, like the Internet 30, the infrastructure of the subnet 10 can become heterogeneous; some nodes 18 can be IPv4 nodes, while others are IPv6 nodes.

In the subnet 10, each node 18 can establish connectivity with one or more other nodes 18 through broadcast or point-to-point links. In general, each link is a communication facility or 15 medium over which nodes 18 can communicate at the link layer (i.e., the protocol layer immediately below the Internet Protocol layer.) Such communication links can be wire-line (e.g., telephone lines) or wireless; thus, nodes 18 are referred to as wireless or wire-line depending upon the type of communication link that the node 18 has to the subnet 10. Examples of wireless communication links are microwave links, radio frequency (RF) links, infrared (IR) 20 links, and satellite links. Protocols for establishing link layer links include Ethernet, PPP (Point-to-Point Protocol) links, X.25, Frame Relay, or ATM (asynchronous transfer mode). Each wireless node 18, e.g., IP host A 12, has a range 22 of communication within which that node 18 can establish a connection to the subnet 10. When beyond the range 22 of communication, the IP host A 12 cannot communicate with the server 40 on the Internet 30 or with other nodes 18 in 25 the subnet 10.

- 9 -

Each broadcast link connecting multiple nodes 18 is mapped into multiple point-to-point bi-directional links. For example, a pair of nodes 18 is considered to have established a bi-directional link 18, if each node 18 can reliably receive messages from the other. For example, IP host A 12 and node B 14 have established a bi-directional link 24 if and only if IP host A 12 can receive messages sent from node B 14 and node B 14 can receive messages sent from IP host A 12 at a given instant in time. Nodes 18 that have established a bi-directional link are considered to be adjacent (i.e., neighboring nodes). Such a bi-directional link 24 between the two nodes A and B is represented by a pair of unidirectional links (A, B) and (B, A). Each link has at least one positive cost (or metric) that can vary in time, and for any given cost, such cost for the link (A, B) may be different from that for the link (B, A). Any technique for assigning costs to links can be used to practice the invention. For example, the cost of a link can be one, for minimum-hop routing, or the link delay plus a constant bias.

In one embodiment, the subnet 10 is a mobile "ad hoc" network ("MANET") in that the topology of the subnet 10 and the state of the links (i.e., link state) between the nodes 18 in the subnet 10 can change frequently because several of the nodes 18 are mobile. That is, each mobile node 18 may move from one location to another location within the same subnet 10 or to another subnet 20, dynamically breaking existing links and establishing new links with other nodes 18, 18' as a result. Such movement by one node 18 does not necessarily result in breaking a link, but may diminish the quality of the communications with another node 18 over that link. In this case, a cost of that link has increased. Movement that breaks a link may interrupt any on-going communications with other nodes 18 in the subnet 10 or in the foreign subnet 20, or with servers (e.g., server 40) connected to the Internet 30. In another embodiment, the position of every node 18 in the subnet 10 is fixed (i.e., a static network configuration in which no link state changes occur due to node mobility). As the principles of the invention apply to both static and

- 10 -

dynamic network configurations, a reference to the subnet 10 contemplates both types of network environments.

The following example illustrates the operation of the subnet 10. Consider, for example, that node A is communicating with the server 40 over a route through subnet 10 that includes the link (A, B) to node B 14, when node A 12 moves from its present location. This movement  
5 breaks the communication link with node B 14 and, as a result, interrupts communications with the server 40. The relocation of node A 12 may break a link with one or more other nodes 18 as well. As one example, the movement by node A 12 may temporarily take node A 12 out of communication range with node B 14, and upon returning within range, node A 12 can  
10 reestablish the broken link 24 with node B 14. In this example, the link 24 is intermittent. As another example, node A 12 may move to a different location within the subnet 10 altogether and reestablish a bi-directional link 26 with a different node, (e.g., here node H). In yet another example, node A 12 may move to the foreign subnet 20 and establish a bi-directional link 28 with a node 14' in the subnet 20 (e.g., node M 14').

15 Each router 14 in the subnet 10 is responsible for detecting, updating, and reporting changes in cost and up-or-down status of each outgoing communication link to neighbor nodes. Thus, each router 14 in the subnet 10 runs a link-state-routing protocol for disseminating subnet topology and link-state information to the other routers 14 in the subnet 10. Each router 14 also executes a neighbor discovery protocol for detecting the arrival of new neighbor nodes and the  
20 loss of existing neighbor nodes. To achieve discovery, IP hosts 12 connected to the subnet 10 also run the neighbor discovery protocol. IP hosts 12 can also operate as routers by running the link-state-routing protocol (in the description, such routing IP hosts are categorically referred to as routers 14). The link-state-routing protocol, referred to as a topology broadcast based on reverse-path forwarding (TBRPF) protocol, seeks to substantially minimize the amount of update



- 11 -

and control traffic required to maintain shortest (or nearly shortest) paths to all destinations in the subnet 10.

In brief, the TBRPF protocol performed by each of the routers 14 in the subnet 10 operates to inform a subset of the neighboring routers 14 in the subnet 10 of the current network topology and corresponding link-state information. Thus, for the examples above, each router 14 in the subnet 10 that detects a change in a link to node A 12, (e.g., node B 14 in the cost of the link (B, A)), operates as the source (i.e., source node) of an update. Each source node sends a message to a neighbor of that source node, informing the neighbor of the update to that link. Each router 14 receiving the update may subsequently forward the update to zero or more neighbor nodes, until the change in the topology of the subnet 10 disseminates to the appropriate routers 14 in the subnet 10.

To transmit update messages, the TBRPF protocol supports unicast transmissions (e.g., point-to-point or receiver directed), in which a packet reaches only a single neighbor, and broadcast transmissions, in which a single packet is transmitted simultaneously to all neighbor nodes. In particular, the TBRPF protocol allows an update to be sent either on a common broadcast channel or on one or more unicast channels, depending on the number of neighbors that need to receive the update.

Upon recovering the same link to node B 14, or upon reestablishing a new link to another node 18 in the same subnet 10 or in the foreign subnet 20, the node A 12 can resume the interrupted communications with server 40. In effect, one of the nodes 18, 18' in the subnet 10, 20, respectively, using the neighbor discovery protocol, discovers node A 12 and, using the TBRPF protocol, initiates dissemination of topology and link-state information associated with the link to node A 12. The routers 14 also use the TBRPF protocol to disseminate this information to the other routers in the respective subnet 10 so that one or more routes to the node A 12 become available for communication with the server 40.

- 12 -

In one embodiment, such communications resume at their point of interruption. In brief, node A 12 maintains, in local cache, copies of objects that are located on the server 40. When node A 12 and the server 40 are in communication, node A 12 updates the objects as necessary, thereby maintaining substantially up-to-date copies of the objects. Thus, when node A 12 moves  
5 out of the communication range 22 with the subnet 10, the node A 12 initially has up-to-date information. Then when node A 12 reconnects to the subnet 10, the server 40 forwards previously undelivered updates to the objects locally stored at node A 12, along a route determined by information stored at each of the routing nodes 14. In the event node A 12 reconnects to the foreign subnet 20, a hand-off protocol, such as MobileIP, is used to achieve the  
10 redirection of the messages between the server 40 and the node A 12.

The route taken by the object updates may traverse a heterogeneous IPv6/IPv4 infrastructure. Normally, IPv6 nodes are unable to route packets to other IPv6 nodes 18 over routes that pass through IPv4 infrastructure. In one embodiment, described in more detail below, the nodes 18 use an IPv6-IPv4 compatible aggregatable global unicast address format to achieve  
15 such routing. This IPv6-IPv4 compatibility address format also enables incremental deployment of IPv6 nodes 18 that do not share a common multiple access data-link with another IPv6 node 18.

Accordingly, the internetworking system 2 provides various mobile ad hoc extensions to the Internet 30 that are particularly suited to the dynamic environment of mobile ad hoc  
20 networks. Such extensions, which are described further below, include techniques for (1) disseminating update information to nodes 18 in the subnet 10 using the TBRPF protocol; (2) detecting the appearance and disappearance of new neighbor nodes using a neighbor discovery protocol; (3) establishing an address format that facilitates deployment of IPv6 nodes in a predominantly IPv4 network infrastructure; (4) updating information upon resuming

- 13 -

communications between nodes; and (5) adaptively using network bandwidth to establish and maintain connections between nodes 18 and the server 40.

Fig. 2 shows a portion of an embodiment of protocol stack 50 that can be used by each of the routing nodes 14, 14' to communicate with other routing nodes 14 in the subnet 10, 20 and on the Internet 30, and thereby implement the various extensions to the Internet 30 described herein. The protocol stack 50 includes a data-link layer 54, a network layer 62, and a transport layer 70.

The data link layer 54 can implemented by any conventional data link protocol (e.g., IEEE 802.11) with an addressing scheme that supports broadcast, multicast and unicast addressing with best-effort (not guaranteed) message delivery services between nodes 18 having instantaneous bi-directional links. For such implementations, each node 18 in the subnet 10 has a unique data link layer unicast address assignment.

The network layer 62 is the protocol layer responsible for assuring that packets arrive at their proper destination. Some of the mobile ad hoc extensions for the Internet 30 described herein operate at the network layer 62, such as the TBRPF protocol 58 and the IPv6-IPv4 compatibility address format, described in more detail below. Embodiments that redirect communications from foreign subnets to home subnets also use hand-off mechanisms such as Mobile IP, which operate at the network layer 62. At the transport layer 70, other mobile ad hoc extensions to the Internet 30 are implemented, such as techniques for updating communications upon restoring connections between nodes and for adaptively using the network bandwidth.

#### **1. Topology Broadcast based on Reverse-Path Forwarding (TBRPF) Protocol**

In brief, the TBRPF protocol uses the concept of reverse-path forwarding to broadcast each link-state update in the reverse direction along a tree formed by the minimum-hop paths from all routing nodes 14 to the source of the update. That is, each link-state update is broadcast

- 14 -

along the minimum-hop-path tree rooted at the source (i.e., source node "src") of the update.

The minimum-hop-path trees (one tree per source) are updated dynamically using the topology and link-state information that are received along the minimum-hop-path trees themselves. In one embodiment, minimum-hop-path trees are used because they change less frequently than  
5 shortest-path trees that are determined based on a metric, such as delay. Other embodiments of the TBRPF protocol can use other types of trees, such as shortest path trees, to practice the principles of the invention.

Based on the information received along the minimum-hop-path trees, each node 18 in the subnet 10 computes a parent node and children nodes, if any, for the minimum-hop-path tree  
10 rooted at each source node src. Each routing node 14 may receive and forward updates originating from a source node src along the minimum-hop-path tree rooted at that source node src. Each routing node 14 in the subnet 10 also engages in neighbor discovery to detect new neighbor nodes and the loss of existing neighbor nodes. Consequently, the routing node 14 may become the source of an update and thus may generate an update message. When forwarding  
15 data packets to a destination node, each routing node 14 selects the next node on a route to the destination.

To communicate according to the TBRPF protocol, each routing node 14 (or node i, when referred to generally) in the subnet 10 stores the following information:

1. A topology table, denoted  $TT_i$ , consisting of all link-states stored at node i. The  
20 entry for link (u, v) in this table is denoted  $TT_i(u, v)$  and includes the most recent update (u, v, c, sn) received for link (u, v). The component c represents the cost associated with the link, and the component sn is a serial number for identifying the most recent update affecting link (u, v) received by the node i. The components c and sn of the entry for the link (u, v) is denoted  $TT_i(u, v).c$  and  $TT_i(u, v).sn$ . Optionally, the dissemination of

- 15 -

multiple link metrics is attainable by replacing the single cost  $c$  with a vector of multiple metrics.

2. A list of neighbor nodes, denoted  $N_i$ .

3. For each node  $u$  other than node  $i$ , the following is maintained:

5 a. The parent, denoted  $p_i(u)$ , which is the neighbor node ("nbr") of node  $i$  that is the next node on a minimum-hop path from node  $i$  to node  $u$ , as obtained from the topology table  $TT_i$ .

b. A list of children nodes of node  $i$ , denoted  $children_i(u)$ .

10 c. The sequence number of the most recent link-state update originating from node  $u$  received by node  $i$ , denoted  $sn_i(u)$ . The sequence number is included in the link-state update message. The use of sequence numbers helps achieve reliability despite topology changes, because node  $i$  avoids sending another node information that the other node already has. Each node  $i$  maintains a counter (i.e., the sequence number) for each link that the node  $i$  monitors. That counter is  
15 incremented each time the status of the link changes.

d. The routing table entry for node  $u$ , consisting of the next node on a preferred path to node  $u$ . The routing table entry for node  $u$  can be equal to the parent  $p_i(u)$  if minimum-hop routing is used for data packets. However, in general, the routing table entry for node  $u$  is not  $p_i(u)$ , because the selection of  
20 routes for data traffic can be based on any objective.

One embodiment of the TBRPF protocol uses the following message types:

LINK-STATE UPDATE: A message containing one or more link-state updates  
( $u, v, c, sn$ ).

NEW PARENT: A message informing a neighbor node that the node has selected  
25 that neighbor node to be a parent with respect to one or more sources of updates.

- 16 -

CANCEL PARENT: A message informing a neighbor that it is no longer a parent with respect to one or more sources of updates.

HELLO: A message sent periodically by each node *i* for neighbor discovery.

NEIGHBOR: A message sent in response to a HELLO message.

5 NEIGHBOR ACK: A message sent in response to a NEIGHBOR message.

ACK: A link-level acknowledgment to a unicast transmission.

NACK: A link-level negative acknowledgment reporting that one or more update messages sent on the broadcast channel were not received.

10 RETRANSMISSION OF BROADCAST: A retransmission, on a unicast channel, of link-state updates belonging to an update message for which a NACK message was received.

HEARTBEAT: A message sent periodically on the broadcast channel when there are no updates to be sent on this channel, used to achieve reliable link-level broadcast of update messages based on NACKs.

15 END OF BROADCAST: A message sent to a neighbor over a unicast channel, to report that updates originating from one or more sources are now being sent on the unicast channel instead of the broadcast channel.

The formats for the various types of TBRPF protocol messages are described below.

#### **Building the Minimum-Hop-Path Tree For a Source**

20 Fig. 3 shows an embodiment of a process by which each routing node 14 selects a parent neighbor node and children neighbor node(s) for each potential source node *src* in the subnet 10. The selection of the parent and children neighbor nodes for each potential source node *src* define a minimum-hop-path tree for that potential source node along which the routing nodes 14 receive and forward link-state updates originating from that source node *src*. Pseudo-code describing the  
25 network-level procedures performed by each routing node 14 is in Appendix A.

- 17 -

Node *i* receives (step 90) a message over a communication link. The received message can represent a link-state update, the discovery of a new neighbor node, the loss of a neighbor node, a change in the cost of a link to a neighbor node, a selection of a new parent neighbor node, or a cancellation of a parent neighbor node. Pseudo-code for processing these types of received messages is provided in Appendix A; the corresponding procedures are called Process\_Update, Link\_Up, Link\_Down, Link\_Change, Process\_New\_Parent, and Process\_Cancel\_Parent, respectively. The general procedure followed in response to all of these events, and the specific procedure followed by a node that has just started and has no topology information are described below.

10        If node *i* receives a message representing a link-state update, the discovery of a new neighbor node, the loss of a neighbor node, or a change in the cost of a link to a neighbor node, node *i* enters (step 100) the new link-state information, if any into the topology table, TT<sub>*i*</sub>, and forwards (step 102) the link-state information in a link-state update to the neighbor nodes in children<sub>*i*</sub>(src), where src is the source node at which the update originated. Node *i* then  
15        computes (step 104) the parent nodes p<sub>*i*</sub>(*u*) for all potential source nodes src by running a shortest-path algorithm such as Dijkstra's algorithm. If this computation results in a change to the parent node p<sub>*i*</sub>(*u*) for any source *u*, node *i* then sends a NEW PARENT(*u*, sn) message, where sn = sn<sub>*i*</sub>(*u*), to the new parent node p<sub>*i*</sub>(*u*) and a CANCEL PARENT message to the old parent node (step 106).

20        If node *i* receives (step 90) a NEW PARENT(*u*, sn) message from a sending node with source *u* and sequence number sn, node *i* adds (step 108) the sending node to node *i*'s list of children nodes children<sub>*i*</sub>(*u*) for that source *u*, and then sends (step 110) the sending node a LINK-STATE UPDATE message containing all updates in node *i*'s topology table, TT<sub>*i*</sub>, originating from source *u* and having a sequence number greater than sn. If node *i* receives (step



- 18 -

90) a CANCEL PARENT(u) message from a sending node with source u, node i removes (step 112) the sending node from node i's list of children nodes  $children\_i(u)$  for that source u.

Consider, for example, the case in which node i initially has no topology information. Accordingly, node i has no links to neighbor nodes, and its topology table  $TT\_i$  is empty. Also  
5 the parent node is  $p\_i(src) = \text{NULL}$  (i.e., not defined), the  $children\_i(src)$  is the empty set, and  $sn\_i(src) = 0$  for each source node src. Upon receiving (step 90) messages representing the discovery of neighbor nodes, node i executes the Link\_Up procedure to process each link established with each neighbor node nbr. Because each neighbor node nbr of node i is (trivially) the next node on the minimum-hop path from node i to neighbor node nbr, node i selects (step  
10 104) each of its neighbor nodes nbr as the new parent node  $p\_i(nbr)$  for source node nbr. Execution of the Link-Up procedure results in node i sending (step 106) a NEW PARENT message to each neighbor node nbr. Therefore, the NEW PARENT message sent to a new neighbor node nbr contains the neighbor node nbr (and possibly other sources) in its source list.

In response to the NEW PARENT message, then each neighbor node nbr informs (step  
15 110) node i of the outgoing links of neighbor node nbr. Information about the outgoing links of neighbor node nbr allows node i to compute minimum-hop paths to the nodes at the other end of the outgoing links, and thus to compute (step 104) new parents  $p\_i(src)$ , for all source nodes src that are two hops away. Node i sends (step 106) a NEW PARENT message to each of these computed new parents. Then each parent  $p\_i(src)$  for each such source node src informs (step  
20 110) node i of the outgoing links for source node src, which allows node i to compute (step 104) new parents for all source nodes that are three hops away. This process continues until node i has computed parent nodes for all source nodes src in the subnet 10. As a result, for a given source node src, the parents  $p\_i(src)$  for all nodes i other than source node src define a minimum-hop-path tree rooted at source node src (after the protocol has converged).

- 19 -

Node  $i$  cancels an existing parent  $p_i(\text{src})$  by sending a CANCEL PARENT(src) message containing the identity of the source node src. Consequently, the set of children,  $\text{children}_i(\text{src})$ , at node  $i$  with respect to source node src is the set of neighbor nodes from which node  $i$  has received a NEW PARENT message containing the identity of source node src without receiving a subsequent CANCEL PARENT message for that source node src. Node  $i$  can also simultaneously select a neighbor node as the parent for multiple sources, so that the node  $i$  sends a NEW PARENT(src\_list, sn\_list) message to the new parent, where src\_list is the list of source nodes and sn\_list is the corresponding list of sequence numbers. Similarly, a CANCEL PARENT message can contain a list of sources.

10 In one embodiment, the TBRPF does not use NEW PARENT and CANCEL PARENT messages in the generation the minimum-hop-path tree. Instead, each node  $i$  computes the minimum-hop paths from each neighbor node nbr to all destinations (e.g., by using breadth-first search or Dijkstra's shortest-path algorithm). Consequently, each node  $i$  computes the parents  $p_{\text{nbr}}(\text{src})$  for each neighbor node nbr and source node src, from which node  $i$  determines which  
15 neighbor nodes nbr are its children for the given source node src. Although this process eliminates NEW PARENT and CANCEL PARENT messages, the process also requires that each node  $i$  (1) sends all updates originating from the source node src to any child node in  $\text{children}_i(\text{src})$ , or (2) periodically sends updates along the minimum-hop-path tree, because node  $i$  does not know the sequence number  $\text{sn}_{\text{nbr}}(\text{src})$  from the neighbor node nbr and thus  
20 does not know what updates the neighbor node nbr already has. Either of these actions ensures that each neighbor node nbr receives the most recent information for each link.

Fig. 4 shows an embodiment of an exemplary minimum-hop-path tree 120 for the nodes 18 in the subnet 10 of Fig. 1. For the sake of illustration, assume that node D is the source of an update. The parent 122 for nodes C, G, and H with respect to the source node D is node D; the  
25 parent 124 for node F with respect to source node D is node H; the parent 126 for nodes A and B

- 20 -

with respect to source node D is node F; the parent 128 for node E is node B; and the parent 130 for node L is node A. (In this example, node A is a routing node 14, and thus runs the TBRPF protocol.)

Conversely, the children 132 of node D are nodes C, G, and H; the child 134 of node H is node F; the children 136 of node F are nodes A and B; the child 138 of node B is node E, and the child 140 of node A is node L. As shown, nodes C, E, G, and L are leaf nodes, which, in accordance with the TBRPF protocol, do not have to forward updates originating from the source node D.

#### **Updating the Minimum-Hop-Path Tree**

10 In brief, the TBRPF protocol disseminates link-state updates generated by a source node src along the minimum-hop-path tree rooted at node src and dynamically updates the minimum-hop-path tree based on the topology and link-state information received along the minimum-hop-path tree. More specifically, whenever the topology table TT<sub>i</sub> of node i changes, the node i computes its parent p<sub>i</sub>(src) with respect to every source node src (see the procedure

15 Update\_Parents in Appendix A). The node i computes parents by (1) computing minimum-hop paths to all other nodes using, for example, Dijkstra's algorithm, and (2) selecting the next node on the minimum-hop path to each source node src to be the parent for that source node src (see the procedure Compute\_New\_Parents in Appendix A). The computation of parents occurs when the node i receives a topology update, establishes a link to a new neighbor node, or detects a

20 failure or change in cost of a link to an existing neighbor node.

In one embodiment, node i computes a new parent p<sub>i</sub>(src) for a given source node src even though the path to the source node src through the new parent has the same number of hops as the path to the source node src through the old parent. In another embodiment, the node keeps the old parent node in this event, thus reducing the overhead of the TBRPF protocol. This

- 21 -

embodiment can be implemented, for example, by using the procedure Compute\_New\_Parents2 (given in Appendix A) instead of the procedure Compute\_New\_Parents.

If the parent  $p_i(src)$  changes, node  $i$  sends the message CANCEL PARENT( $src$ ) to the current (i.e., old) parent, if the old parent exists. Upon receiving the CANCEL PARENT( $src$ ) message, the old parent (" $k$ ") removes node  $i$  from the list  $children\_k(src)$ .

Node  $i$  also sends the message NEW PARENT( $src, sn$ ) to the newly computed parent if the new parent exists, where  $sn = sn\_i(src)$  is the sequence number of the most recent link-state update originating from source node  $src$  received by node  $i$ . This sequence number indicates the "position" up to which node  $i$  has received updates from the old parent, and indicates to the new parent that it should send only those updates that occurred subsequently (i.e., after that sequence number).

Upon receiving the NEW PARENT( $src, sn$ ) message, the new parent " $j$ " for  $p_i(src)$  adds node  $i$  to the list  $children\_j(src)$  and sends to node  $i$  a link-state update message consisting of all the link states originating from source node  $src$  in its topology table that have a sequence number greater than  $sn$  (see the procedure Process\_New\_Parent in Appendix A). Thus, only updates not yet known to node  $i$  are sent to node  $i$ .

Generally, the range of sequence numbers is large enough so that wraparound does not occur. However, if a small sequence number range is used, wraparound can be handled by employing infrequent periodic updates with a period that is less than half the minimum wraparound period, and by using a cyclic comparison of sequence numbers. That is,  $sn$  is considered less than  $sn'$  if either  $sn$  is less than  $sn'$  and the difference between  $sn$  and  $sn'$  ( $sn' - sn$ ) is less than half the largest possible sequence number, or  $sn'$  is less than  $sn$  and the difference,  $sn - sn'$ , is greater than half the largest possible sequence number.

When a node  $i$  detects the existence of a new neighbor  $nbr$ , it executes Link\_Up( $i, nbr$ ) to process this newly established link. The link cost and sequence number fields for this link in the

- 22 -

topology table at node  $i$  are updated. Then, the corresponding link-state message is sent to all neighbors in  $children\_i(i)$ . As noted above, node  $i$  also recomputes its parent node  $p\_i(src)$  for every node  $src$ , in response to this topological change. In a similar manner, when node  $i$  detects the loss of connectivity to an existing neighbor node  $nbr$ , node  $i$  executes  $Link\_Down(i, nbr)$ .

- 5  $Link\_Change(i, nbr)$  is likewise executed at node  $i$  in response to a change in the cost to an existing neighbor node  $nbr$ . However, this procedure does not recompute parents.

In one embodiment, if a path between the node  $i$  and a given source node  $src$  ceases to exist, the node  $i$  computes a new parent  $p\_i(src)$  that is set to NULL (i.e., parent does not exist). In another embodiment, although the path between the node  $i$  and the given source node  $src$  ceases to exist, the node  $i$  keeps the current parent, if the current parent is still a neighbor node of the node  $i$ . Thus, the overhead of the TBRPF protocol is reduced because it is unnecessary to send a CANCEL PARENT and a subsequent NEW PARENT messages if the old path to the source becomes operational later because of a link recovery. This embodiment can be implemented by replacing the fifth line of the procedure  $Update\_Parents$  in Appendix A, “If (new\_p\_i(src) != p\_i(src)) {”, with the line “If (new\_p\_i(src) != p\_i(src) and new\_p\_i(src) != NULL) {”.

10  
15

The TBRPF protocol does not use an age field in link-state update messages. However, failed links (represented by an infinite cost) and links that are unreachable (i.e., links  $(u, v)$  such that  $p\_i(u) = NULL$ ) are deleted from the topology table  $TT\_i$  after  $MAX\_AGE$  seconds (e.g., 1 hour) in order to conserve memory. Failed links  $(u, v)$  are maintained for some time in the topology table  $TT\_i$ , rather than deleted immediately, to ensure that the node  $i$  that changes its parent  $p\_i(u)$  near the time of failure (or had no parent  $p\_i(u)$  during the failure) is informed of the failure by the new parent.

20

Unreachable links, (i.e., links  $(u, v)$  such that node  $i$  and node  $u$  are on different sides of a network partition), are maintained for a period of time to avoid having to rebroadcast the old link

25

- 23 -

state for (u, v) throughout node i's side of the partition, if the network partition soon recovers, which can often happen if the network partition is caused by a marginal link that oscillates between the up and down states. If a link recovers resulting in the reconnection of two network components that were disconnected (i.e., partitioned) prior to the link recovery, the routing nodes  
5 14 in one partition may temporarily have invalid routes to nodes 18 in the other partition. This occurs because the routing nodes 14 may receive an update message for the link recovery before receiving update messages for links in the other partition. Consequently, the link-information for those links in the other partition may be outdated temporarily.

To correct this situation, in one embodiment, a header field is added to each link-state  
10 update message, which indicates whether the update message is sent in response to a NEW PARENT message. The header field also identifies the corresponding NEW PARENT message using a sequence number. For example, if a given node i sends a NEW PARENT message (for multiple sources) to node j following the recovery of the link (i, j), the node i waits for a response from node j to the NEW PARENT message before sending to node i's neighbor nodes  
15 an update message corresponding to the link recovery. The response from node j includes the link-state information of the other nodes 18 in the previously disconnected partition. Then node i forwards this link-state information to node i's neighbor nodes. Consequently, the nodes 18 in the same partition as node i receives updates for the links in the other partition at the same time that the nodes 18 receive the update for the link recovery. Thus, the link-state information for  
20 those links in the other partition is not outdated temporarily.

A node i that is turned off (or goes to sleep) operates as if the links to all neighbors have gone down. Thus, the node i remembers the link-state information that it had when turned off. Since all such links are either down or unreachable, these link states are deleted from the topology table TT<sub>i</sub> if the node i awakens after being in sleep mode for more than MAX\_AGE  
25 seconds.



- 24 -

Infrequent periodic updates occur to correct errors that may appear in table entries or update messages. (See Send\_Periodic\_Updates in Appendix A.) As discussed above, periodic updates are also useful if the sequence number range is not large enough to avoid wraparound.

### Initiating an Update Message

5        When a given routing node 14 detects a change in the state of a neighbor node, that routing node 14 becomes the source (i.e., source node src) of a link-state update message with respect to corresponding link to that neighbor node. As described above, the source node src then broadcasts each link-state update along the minimum-hop-path tree rooted at the source of the update.

10        A link-state update message reports the state of the link (src, nbr) as a tuple (src, nbr, c, sn), where c and sn are the cost and the sequence number associated with the update. A cost of infinity represents a failed link. The source node src is the head node of link (src, nbr), and is the only node that can report changes to parameters of link (src, nbr). Therefore, any node 18 receiving the link-state update (src, nbr, c, sn) can determine that the update originated from the  
15        source node src.

      The source node src maintains a counter sn\_src, which is incremented by at least one each time the cost of one or more outgoing links (src, nbr) changes value. For example, the counter sn\_src can be a time stamp that represents the number of seconds (or other units of time) elapsed from some fixed time. When the source node src generates a link-state update  
20        (src, nbr, c, sn), the sequence number sn is set to the current value of sn\_src.

### Receiving an Update Message

      In brief, each routing node 14 that receives a link-state update message receives that update message along a single path. That is, any link-state update originating from source node src is accepted by node i if (1) the link-state update is received from the parent node p\_i(src), and  
25        (2) the link-state update has a larger sequence number than the corresponding link-state entry in



- 25 -

the topology table TT<sub>i</sub> at node i. If the link-state update is accepted, node i enters the link-state update into the topology table TT<sub>i</sub>. Node i may then forward the link-state update to zero or more children nodes in children<sub>i</sub>(src). In one embodiment, the link-state update passes to every child node in children<sub>i</sub>(src). (See the procedures Update\_Topology\_Table and Process\_Update  
5 in the Appendix A.)

### Forwarding Update Messages

In most link-state routing protocols, e.g., OSPF (Opens Shortest Path First), each routing node 18 forwards the same link-state information to all neighbor nodes. In contrast, in one embodiment of the TBRPF protocol, each routing node 14 sends each link-state update only to  
10 neighbor nodes that are children on the minimum-hop-path tree rooted at the source of the update. Each routing node 14 having no children for the source node src of the link-state update is a leaf in the minimum-hop-path tree and therefore does not forward updates originating from the source node src. In typical networks, most nodes 18 are leaves, thus the TBRPF protocol makes efficient use of the bandwidth of the subnet 10. In addition, those nodes having only one  
15 child node for the source node src can send updates generated by the source node src to that child node only, instead of broadcasting the updates to all neighbor nodes.

The TBRPF protocol may utilize bandwidth more efficiently by using unicast transmissions if those routing nodes 14 have only one child, or a few children, for the source of the update, and broadcast transmissions when several children exist for the update. Therefore, in  
20 one embodiment, the TBRPF protocol determines whether to use unicast or broadcast transmissions, depending on the number of children nodes and the total number of neighbor nodes.

In general, each routing node 14 uses unicast transmissions for updates with only one intended receiver (e.g., only one child), and broadcast transmissions for updates with several  
25 intended receivers, to avoid transmitting the update message several times. Therefore, each

- 26 -

routing node 14 uses unicast transmission if  $k = 1$  and use broadcast if  $k > 1$ , where  $k$  is the number of intended receivers. A possible drawback can occur if the number of children nodes exceeds one and there are a many more neighbors. For example, if there are two children nodes and twenty neighbor nodes, (i.e.,  $k = 2$  and  $n = 20$ , where  $k$  is the number of children nodes and  $n$  is the number of neighbors), then 18 neighbor nodes are listening to a message not intended for them. Such neighbor nodes could instead be sending or receiving other messages.

To avoid this possible drawback, one option is to use broadcast transmission if  $k > (n+1)/2$  and unicast transmission in all other cases. In general, a rule of the form  $k > g(n)$  can be used. For update messages, the number of children  $k$  may be different for different update sources. Therefore, it is possible to use unicast transmissions for some sources and broadcast transmissions for other sources, and the transmission mode for a given source  $u$ , denoted  $mode\_i(u)$ , can change dynamically between unicast and broadcast as the number of children changes.

While LINK-STATE-UPDATE messages can be transmitted in either unicast or broadcast mode, HELLO messages and HEARTBEAT messages (discussed below) are always transmitted on the broadcast channel, and the following messages are always transmitted on the unicast channel (to a single neighbor): NEIGHBOR, NEIGHBOR ACK, ACK, NACK, NEW PARENT, CANCEL PARENT, RETRANSMISSION OF BROADCAST, END OF BROADCAST, and LINK-STATE-UPDATE messages sent in response to a NEW PARENT message.

Exemplary pseudo-code for a procedure for sending a LINK-STATE UPDATE message (that is not a response to a NEW PARENT message) on the broadcast or unicast channel is as follows:

If ( $mode\_i(src) = BROADCAST$ )

- 27 -

Append the message update\_msg to the message queue associated with the broadcast channel.

If (mode\_i(src) == UNICAST)

For (each node k in children\_i(src))

5                   Append the message update\_msg to the message queue associated with the unicast channel to node k.

Reliable unicast transmission of control packets can be achieved by a variety of reliable link-layer unicast transmission protocols that use sequence numbers and ACKs, and that retransmit a packet if an ACK is not received for that packet within a specified amount of time.

#### 10   **Reliable Transmission in Broadcast Mode**

For reliable transmission of Link-State Update messages in broadcast mode, each broadcast update message includes one or more link-state updates, denoted lsu(src), originating from sources src for which the transmission mode is BROADCAST. Each broadcast control packet is identified by a sequence number that is incremented each time a new broadcast control packet is transmitted. Reliable transmission of broadcast control packets in TBRPF can be  
15                   accomplished using either ACKs or NACKs. If ACKs are used, then the packet is retransmitted after a specified amount of time if an ACK has not been received from each neighbor node that must receive the message.

In one embodiment of TBRPF, NACKs are used instead of ACKs for reliable  
20                   transmission of broadcast control packets, so that the amount of ACK/NACK traffic is minimized if most transmissions are successful. Suppose node i receives a NACK from a neighbor node nbr for a broadcast update message. In one embodiment, all updates lsu(src) in the original message, for each source node src such that neighbor node nbr belongs to children\_i(src), are retransmitted (reliably) on the UNICAST channel to the neighbor node nbr,  
25                   in a RETRANSMISSION OF BROADCAST message. This message includes the original broadcast sequence number to allow neighbor node nbr to process the updates in the correct

- 28 -

order. In another embodiment, such update messages are retransmitted on the broadcast channel. This embodiment may improve the efficiency of the TBRPF protocol in subnets that do not support receiver-directed transmission, because in such subnets unicast transmission provides no efficiency advantage over broadcast transmissions.

5           The procedure for the reliable transmission of broadcast update packets uses the following message types (in addition to LINK-STATE UPDATE messages): HEARTBEAT(sn), NACK(sn, bit\_map), and RETRANSMISSION OF BROADCAST(sn, update\_msg). A NACK(sn, bit\_map) message contains the sequence number (sn) of the last received broadcast control packet, and a 16-bit vector (bit\_map) specifying which of the 16 broadcast control  
10   packets from sn-15 to sn have been successfully received.

A description of the procedure for the reliable transmission of broadcast update packets at node i uses the following exemplary notation:

Pkt(sn) represents a control packet with sequence number sn transmitted on the broadcast channel by node i.  
15   MsgQ represents a message queue for new control messages to be sent on the broadcast channel from node i.  
brdcst\_sn\_i represents the sequence number of the last packet transmitted on the broadcast channel by node i.  
Heartbeat\_Timer represents a timer used in the transmission of the HEARTBEAT  
20   message.

Following the transmission of the broadcast control packet Pkt(brdcst\_sn\_i) on the broadcast channel, node i increments brdcst\_sn\_i and reinitializes Heartbeat\_Timer. When Heartbeat\_Timer expires at node i, the node i appends the control message HEARTBEAT(brdcst\_sn\_i) to the message queue associated with the broadcast channel, and

- 29 -

reinitializes Heartbeat\_Timer. When the node i receives NACK(sn, bit\_map) from neighbor node nbr, node i performs the functions as illustrated by following exemplary pseudo-code:

```

For each (sn' not received as indicated by bit_map){
    Let update_msg = {(src*, v*, sn*, c*) in Pkt(sn')} such that the neighbor
5      node nbr is in children_i(src*)}.
    Append the message RETRANSMISSION OF
      BROADCAST(sn', update_msg) to the message queue associated
      with the unicast channel to neighbor node nbr. (Message must be
      sent even if update_msg is empty.)}

```

10        Upon receipt at neighbor node nbr of control packet Pkt(sn) transmitted on the broadcast channel by node i, the neighbor node nbr performs the following operations as illustrated by the following pseudo-code:

```

If the control packet Pkt(sn) is received in error{
    Append the control message NACK(sn, bit_map) to the message queue
15      associated with the unicast channel to node i.}
    If the control packet Pkt(sn) is received out of order (i.e., at least one
      previous sequence number is skipped){
        Withhold the processing of the control packet Pkt(sn).
        Append the control message NACK(sn, bit_map') to the
20      message queue associated with the unicast channel
        to node i.}
    Else (control packet Pkt(sn) is received correctly and in order){
        For each Link-State Update message update_msg in
          Pkt(sn), call Process_Update(i, nbr, update_msg).}

```

25        When a communication link is established from node i to a new neighbor nbr, in one embodiment the node i obtains the current value of brdcst\_sn\_nbr from the NEIGHBOR message or NEIGHBOR ACK that was received from neighbor node nbr.

- 30 -

Each node  $i$  can dynamically select the transmission mode for link-state updates originating from each source node  $src$ . As described above, this decision uses a rule of the form  $k > g(n)$ , where  $k$  is the number of children (for  $src$ ) and  $n$  is the number of neighbors of node  $i$ . However, to ensure that updates are received in the correct order, or that the receiver has enough  
 5 information to reorder the updates, node  $i$  sends an END OF BROADCAST( $last\_seq\_no, src$ ) message on the unicast channel to each child when the mode changes to UNICAST, and waits for all update packets sent on unicast channels to be ACKed on before changing to BROADCAST mode.

To facilitate this process, each node  $i$  maintains a binary variable  $unacked\_i(nbr, src)$  for  
 10 each neighbor node  $nbr$  and source node  $src$ , indicating whether there are any unACKed control packets sent to neighbor node  $nbr$  containing link-state updates originating at source node  $src$ . The following exemplary pseudo-code illustrates an embodiment of a procedure that is executed periodically at each node  $i$ .

```

    For each (node  $src$ ){
15      If ( $mode\_i(src) = BROADCAST$  and  $|children\_i(src)| \leq g(n)$ ){
          For each (node  $nbr$  in  $children\_i(src)$ ){
              Append the message END OF BROADCAST( $brdcst\_sn\_i, src$ ) to
                  the message queue associated with the unicast channel to
                  node  $nbr$ .}
20      Set  $mode\_i(src) = UNICAST$ .}

      If ( $mode\_i(src) = UNICAST$  and  $|children\_i(src)| > g(n)$ ){
          Set  $switch\_flag = YES$ .
          For each (node  $nbr$  in  $children\_i(src)$ ){
              If ( $unacked\_i(nbr, src) = YES$ ) Set  $switch\_flag = NO$ .}
25      If ( $switch\_flag = YES$ ) Set  $mode\_i(src) = BROADCAST$ .}}
  
```

- 31 -

**Full and Partial Topology TBRPF**

In one embodiment, a result of the running the TBRPF protocol is that each router 14 in the subnet 10 obtains the state of each link in the subnet 10 (or within a cluster if hierarchical routing is used). Accordingly, this embodiment of the TBRPF protocol is referred to as full-topology link-state protocol. In some embodiments, described below, the TBRPF protocol is a partial-topology link-state protocol in that each router 14 maintains a subset of the communication links in the subnet 10. In the full-topology protocol embodiment, each routing node 14 is provided with the state of each link in the subnet 10 (or cluster, if hierarchical routing is used). In other embodiments, the TBRPF is a partial topology protocol in that each routing node 14 is provided with only a subset of the links in the subnet 10.

For the full-topology link-state protocol embodiment (1) alternate paths and disjoint paths are immediately available, allowing faster recovery from failures and topology changes; and (2) paths can be computed subject to any combination of quality-of-service (QoS) constraints and objectives. Partial-topology link-state protocols provide each node 18 with sufficient topology information to compute at least one path to each destination. Whether implemented as a full-topology or as a partial-topology protocol, the TBRPF protocol is a proactive link-state protocol in that each node 18 dynamically reacts to link-state and topology changes and maintains a path to each possible destination in the subnet 10 at all times.

**A Partial-Topology Embodiment**

In one partial-topology embodiment, each routing node 14 decides which of its outgoing links (i, j), called "special links," should be disseminated to all nodes in the subnet 10. This subset of links is maintained in a list  $L_i$ . All other outgoing links are sent only one hop (i.e., to all neighbor nodes of node i). Node i sends an update to its neighbor nodes if that update is the addition or removal of a link from the list  $L_i$ , or reflects a change in the state of a link in the list  $L_i$ .



- 32 -

Various rules can be used to define the set of special links in the list  $L_i$ . For example, one rule defines a link  $(i, j)$  to be in  $L_i$  only if node  $j$  is the parent of node  $i$  for some source node other than node  $j$ , or if node  $j$  belongs to the set  $children_i(src)$  for some source node  $src$  other than node  $i$ . This definition of special links includes enough links to provide minimum-hop paths between any pair of nodes. As a result, this partial-topology embodiment reduces the amount of control traffic without reducing the quality of the routes. In this embodiment, an update  $(u, v, c, sn, sp)$  is augmented to include a, "sp" field (e.g., a single-bit field), which indicates whether the link  $(u, v)$  is a special link. Pseudo-code representing an exemplary implementation of the partial-topology embodiment appears in the Appendix A, after the "Partial-Topology 1" header. The procedure  $Mark\_Special\_Links(i)$  is called upon a change to the parent  $p_i(src)$  or to the set of children nodes  $children_i(src)$ .

### A Second Partial-Topology Embodiment

In another partial-topology embodiment, each routing node 14, hereafter node  $i$ , maintains a topology table  $TT_i$ , a source tree  $T_i$  (i.e., computed paths to all destinations), a set of reported links  $R_i$ , and a set of neighbor nodes  $N_i$ . The entry of  $TT_i$  for a link  $(u, v)$  is denoted  $TT_i(u, v)$  and consists of the tuple  $(u, v, c, c')$ , where  $c$  is the cost associated with the link and  $c'$  is the last cost reported to neighbor nodes for the link. The component  $c$  of the entry for link  $(u, v)$  is denoted  $TT_i(u, v).c$ . In addition, a parent  $p_i(u)$  and set of children nodes  $children_i(u)$  are maintained for each node  $u \neq \text{node } i$ . The parent  $p_i(u)$  is the next node on a shortest path to node  $u$ , based on the information in  $TT_i$ . The source tree  $T_i$ , computed by a lexicographic version of Dijkstra's algorithm, is the set of links that belong to at least one of the computed paths. The set of reported links  $R_i$  includes the source tree  $T_i$  and any link in  $TT_i$  for which an update has been sent but a delete update has not since been sent. In addition, a binary variable  $pending_i(u)$  is maintained for each node  $u \neq \text{node } i$ , which indicates that the parent  $p_i(u)$  is pending, i.e., that a NEW PARENT( $u$ ) message has been sent to  $p_i(u)$  but no response

- 33 -

has yet been received. In general, each node  $i$  reports to neighbor nodes the current states of only those links in its source tree  $T_i$ , but sends only part of its source tree  $T_i$  to each neighbor node such that no node receives the same information from more than one neighbor node.

Pseudo-code representing an exemplary implementation of this partial-topology embodiment of the TBRPF protocol appears in the Appendix A, after the "Partial-Topology 2" header.

Upon receiving an update message, consisting of one or more updates  $(u, v, c)$ , node  $i$  executes the procedure `Update()`, which calls the procedure `Update_Topology_Table()`, then executes the procedure `Lex_Dijkstra()` to compute the new source tree  $T_i$  and the procedure `Generate_Updates()` to generate updates and modify the set of reported links  $R_i$  based on changes in link costs and changes to the source tree  $T_i$ . Each generated update is then sent to the appropriate children, that is, updates for links with head  $u$  are sent to `children_i(u)`. The procedure `Update_Parents()` is called, which determines any changes in the parent assignment and sends NEW PARENT and CANCEL PARENT messages.

The sending of updates can be accomplished in different ways, depending on whether the subnet consists of point-to-point links, broadcast links, or a combination of both link types. In a network of point-to-point links, each neighbor node  $k$  would be sent a message that contains the updates for links  $(u, v)$  such that  $k$  belongs to `children_i(u)`. If a broadcast capability also exists, links  $(u, v)$  for which `children_i(u)` has more than one member can be broadcast to all neighbor nodes.

The procedure `Update_Topology_Table()` does the following for each update  $(u, v, c)$  in the input message (`in_message`) such that the parent  $p_i(u)$  is the neighbor node who sent the message. (Updates received from a node other than the parent are ignored.) If either `TT_i` does not contain an entry for  $(u, v)$  or contains an entry with a different cost than  $c$ , then `TT_i(u, v)` is updated with the new value  $c$  and link  $(u, v)$  is marked as changed. If the input message is a PARENT RESPONSE, then in addition to updates, the message contains the same list of sources

- 34 -

as the NEW PARENT message to which it is responding. For each such source node  $u$  such that  
 pending<sub>i</sub>( $u$ ) = 1 and for each link ( $u, v$ ) in TT<sub>i</sub> that is outgoing from source node  $u$  but for  
 which the input message does not contain an update, the cost of ( $u, v$ ) is set to infinity, to  
 indicate that the link should be deleted. In other words, any link that was reported by the old  
 5 parent but is not reported by the new parent is deleted. Only information from the current parent  
 is considered valid.

The procedure Lex\_Dijkstra() (not included in Appendix A) is an implementation of  
 Dijkstra's algorithm that computes the lexicographically smallest shortest path LSP( $i, u$ ) from  
 node  $i$  to each node  $u$ , using as path name the sequence of nodes in the path in the reverse  
 10 direction. For example, the next-to-last node of LSP( $i, u$ ) has the smallest node ID among all  
 possible choices for the next-to-last node. Such paths are computed using a modification of  
 Dijkstra's algorithm in which, if there are multiple choices for the next node to label, the one  
 with the smallest ID is chosen.

The procedure Generate\_Updates() decides what updates to include in the message to be  
 15 sent to neighbor nodes. A non-delete update is included for any link ( $u, v$ ) that is in the new  
 source tree  $T_i$  and either is marked as changed or was not in the previous source tree (denoted  
 old source tree  $T_i$ ). In this case,  $T_i(u, v).c'$  is set to  $T_i(u, v).c$ , and ( $u, v$ ) is added to the reported  
 link set  $R_i$  if not already in the reported link set  $R_i$ . A delete update is included for any link ( $u,$   
 $v$ ) that is in the reported link set  $R_i$  but is not in the source tree  $T_i$ , such that  $TT_i(u,$   
 20  $v).c > TT_i(u, v).c'$ . In this case, ( $u, v$ ) is removed from the reported link set  $R_i$ . Any links with  
 infinite cost are erased from the topology table TT<sub>i</sub>.

The procedure Update\_Parents() sets the new parent  $p_i(u)$  for each source node  $u$  to be  
 the second node on the shortest path to node  $u$ . If there is no path to node  $u$ ,  $p_i(u)$  is null. If the  
 new parent is different from the old parent, then a NEW PARENT message is sent to the new  
 25 parent (if it is not null) and a CANCEL PARENT message is sent to the old parent (if it is not

- 35 -

null and the link to the old parent is still up). The NEW PARENT messages for all source nodes  $u$  having the same new parent are combined into a single message, and CANCEL PARENT messages are similarly combined.

The procedure `Process_New_Parent()` is executed when a NEW PARENT message is  
5 received from some neighbor node. For each source node  $u$  in the NEW PARENT message, the procedure adds the neighbor node to `childreni(u)` and includes in the PARENT RESPONSE message an update for each link  $(u, v)$  in the source tree  $T_i$  whose head is source node  $u$ , if such a link exists. (Such a link will not exist if node  $u$  is a leaf of source tree  $T_i$ .) As described above, the PARENT RESPONSE also includes the same list of sources as the NEW PARENT  
10 message to which it is responding. (This list is not necessary if the node sending the NEW PARENT message remembers the list and can match the PARENT RESPONSE to the NEW PARENT message.)

When the cost of a link to a neighbor node  $j$  changes, node  $i$  sets `TTi(i, j).c` to the new cost and calls the procedure `Update()` with  $k = i$  and an empty input message. A threshold rule  
15 can be used so that `TTi(i, j).c` is updated only if the percent difference between the new cost and the old cost is at least some given threshold. If a link to a neighbor node  $j$  fails, the same procedure is followed (with the cost changing to infinity), and node  $j$  is removed from set of neighbor nodes  $N_i$ .

When a link to a neighbor node  $j$  comes up, either initially or upon recovering from a  
20 failure, node  $i$  executes the procedure `Link_Up(I, j)`, which adds neighbor node  $j$  to the set of neighbor nodes  $N_i$ , sets `TTi(i, j).c` to the link cost, and calls the procedure `Update()` with  $k = i$  and an empty input message. This may result in a NEW PARENT message being sent to neighbor node  $j$ .

To correct errors that may appear in `TTi` due to noisy transmissions or memory errors,  
25 each node  $i$  can periodically generate updates for its outgoing links. Since a received update is

- 36 -

ignored unless it has a cost that differs from the entry in the topology table TT<sub>i</sub>, the cost of the periodic update should be chosen to be slightly different from the previous update.

Alternatively, each update can contain an additional bit b, which toggles with each periodic update.

5           Fig. 5 illustrates the operation of the second partial-topology embodiment of the TBRPF protocol when a communication link 142 between nodes B and D in the subnet 10 fails. The minimum-hop-path tree for source node B before the link failure is shown with solid arrows; the minimum-hop-path tree for source node C is shown with dashed arrows. As shown node A selects node B as parent for source nodes B, D, and F, and selects node C as parent for source  
10 nodes C, E, and F. Therefore, node B reports link-state changes to node A only for links (B, A), (B, C), (B, D), and (D, F), and node C reports link-state changes to node A only for links (C, A), (C, B), (C, E), and (E, G). Neither nodes B or C would report a link-state change affecting link (F, G) to node A. Thus, unlike the full-topology embodiment of the TBRPF, in which each node  
14 has link information for every link in the subnet 10, the nodes 18 of this partial-topology  
15 embodiment have link-state information for less than every link in the subnet 10.

          If link (B, D) fails, as shown in Fig. 5, node B reports to nodes A and C that link (B, D) has failed (cost = infinity). Node C reports to node A that link (E, D) 144 has been added to node C's minimum-hop-path source tree. After receiving these updates, node A selects node C as its new parent for source nodes D and F, and sends a NEW PARENT message to node C and a  
20 CANCEL PARENT message to node B. Node C responds by sending node A an update only for link (D, F), because link (D, F) is the only link in node C's minimum hop-path source tree with node D or node F as the head of a link. For example, node F is the head of the link (F, G), but the link (F, G) is not in node C's minimum-hop-path source tree and is therefore not reported to node A. Although the minimum-hop-path source tree of node A is modified during the update

- 37 -

process, node A does not generate any updates because it has no children for any source other than itself (i.e., node A).

### **TBRPF Protocol Messages**

To disseminate link-state updates to the appropriate nodes in the subnet 10, neighboring router nodes 14 that have established bi-directional links and performed data link to IPv4 address resolution using TBRPF neighbor discovery (as described below) exchange TBRPF protocol messages. The IPv4 addresses are therefore available for use as node IDs in TBRPF protocol messages.

In one embodiment, the TBRPF protocol messages are sent via the User Datagram Protocol (UDP), which requires an official UDP-service port-number registration. The use of UDP/IPv4 provides several advantages over a data link level approach, including (1) IPv4 segmentation/reassembly facilities, (2) UDP checksum facilities, (3) simplified application level access for routing daemons, (4) IPv4 multicast addressing for link state messages.

TBRPF protocol messages are sent to the IPv4 unicast address of a current neighbor or to the "All\_TBRPF\_Neighbors" IPv4 multicast address, presuming that an official IPv4 multicast address is assigned to "All\_TBRPF\_Neighbors." In general, a message is sent to the IPv4 unicast address of a current neighbor node if all components of the message pertain only to that neighbor. Similarly, a message is sent to the All\_TBRPF\_Neighbors IPv4 multicast address if the message contains components which pertain to more than one neighbor neighbors. Nodes 14 are prepared to receive TBRPF protocol messages sent to their own IPV4 unicast address or the All\_TBRPF\_Neighbors multicast address.

Actual addressing strategies depend on the underlying data link layer. for example, for data links such as IEEE 802.11, a single, multiple access channel is available for all unicast and broadcast/multicast messages. In such cases, since channel occupancy for unicast and multicast messages is identical, it is advantageous to send a single message to the All\_TBRPF\_Neighbors



multicast address rather than multiple unicast messages, even if the message contains components that pertain to only a subset of the current neighbor nodes. In other cases, in which point-to-point receiver directed channels are available, sending multiple unicast messages may reduce contention on the multiple access broadcast channel.

5    **Atomic TBRPF Message Format**

Fig. 6 shows an exemplary embodiment of an individual (atomic) TBRPF protocol message 160 including a message header 162 followed by a message body 164. Atomic messages may be transmitted either individually or as components of a compound TBRPF protocol message having multiple atomic messages within a single UDP/IPv4 datagram. TBRPF message headers 162 are either 32-bits or 64-bits in length depending on whether the atomic message is BROADCAST or UNICAST.

The message header 162 includes a type field 166, a version field 168, a mode field 170, a number of sources field 172, an offset field 174, a link sequence number field 176, and a receiver identification field 178, which is used when the mode is defined as UNICAST.

The type field 166 (e.g., 4 bits) represents the atomic message type. The following are examples of atomic message types:

	ACK	1	
	NACK	2	
	NEW_PARENT		3
20	CANCEL_PARENT	4	
	HEARTBEAT	5	
	END_OF_BROADCAST	6	
	LINK_STATE_UPDATE_A	7	
	LINK_STATE_UPDATE_B	8	
25	RETRANSMISSION_OF_BROADCAST		9

The version field 168 (e.g., 3 bits) represents the TBRPF protocol version and provides a transition mechanism for future versions of the TBRPF protocol. Also, the version 168 can assist the node 18 in identifying false messages purporting to be TBRPF protocol messages.

- 39 -

The mode field 170 (e.g., 1 bit) represents the transmission mode for the atomic TBRPF protocol message 160; the mode is either UNICAST or BROADCAST. UNICAST refers to an atomic message that must be processed by only a single neighbor node. BROADCAST refers to an atomic message that is to be processed by all neighbor nodes. (For IPv4 subnets, UNICAST implies a specific IPv4 unicast address, whereas BROADCAST implies the All\_TBRPF\_Neighbors IPv4 multicast address.) The following exemplary mode bits are defined:

UNICAST	0
BROADCAST	1

10 Messages of type ACK, NACK, NEW\_PARENT, CANCEL\_PARENT, RETRANSMISSION\_OF\_BROADCAST, and END\_OF\_BROADCAST are sent as UNICAST. Messages of type LINK\_STATE\_UPDATE\_A and LINK\_STATE\_UPDATE\_B may be sent as either UNICAST or BROADCAST.

The number of sources field 172 (e.g., 8 bits) represents the number of sources  
15 "Num\_Sources" included in the atomic message 160. The field 172 takes a value from 1 to 255 for messages of type: NEW\_PARENT, CANCEL\_PARENT, LINK\_STATE\_UPDATE\_A, and LINK\_STATE\_UPDATE\_B. All other message types are set Num\_Sources = 0.

The offset field 174 (e.g., 18 bits) represents the offset (in bytes) from the 0'th byte of the current atomic message header 162 to the 0'th byte of the next atomic message header 162 in the  
20 "compound message" (described below.) An offset of 0 indicates that no further atomic messages follow. The 18-bit offset field 174, for example, imposes a 4-kilobyte length restriction on individual atomic messages.

The sequence number field 176 (e.g., 4 bits) represents the link sequence number ("LSEQ") for this TBRPF protocol message 160.

- 40 -

The receiver identification field 178 (e.g., 32 bits) represents the IPv4 address of the receiving node which is to process this atomic message 160. All nodes 18 other than the node identified by the identification field 178 do not process this atomic message 160. This field 178 is used only if the mode field 170 is set to UNICAST.

## 5 **Compound TBRPF Protocol Message Format**

Fig. 7 shows the format for a compound TBRPF protocol message 180, which includes multiple (i.e., "N") atomic TBRPF messages 160, 160', 160'' that are concatenated to form the compound message 180 within a single UDP/IPv4 packet. Atomic message headers 162, in one embodiment, are aligned on 32-bit boundaries, therefore an atomic message body 164 with a non-integral number of 32-bit words includes 1, 2 or 3 padding bytes 182, 182' preceding a subsequent message header 162', 162'', respectively.

### **TBRPF Atomic Message Body Format**

The format of the atomic message body 164 depends on the value in the type field 166 in the corresponding message header 162. The following are exemplary formats for an atomic message body 164.

#### ACK

The ACK message carries a NULL message body. A 4-bit acknowledgment sequence number (from 0...15) is carried in the LSEQ field 176 of the TBRPF message header 162.

#### NACK:

Each NACK message is a 16-bit vector. Each bit indicates whether each of the last 16 messages prior to the 4-bit sequence number supplied in the LSEQ field 176 of the TBRPF message header 162 was received or lost. As described above, the LSEQ field 176 is set to the sequence number of the last broadcast message received from the neighbor node to which the NACK is being sent.

- 41 -

NEW PARENT:

Fig. 8A shows an embodiment of an exemplary format 186 for a NEW PARENT message. The format 186 includes one or more source node identity fields 188, 188', 188'' (generally 188) and one or more corresponding sequence number fields 190, 190', 190''

5 (generally 190). Each source node identity field 188 holds a value (e.g., 32 bits) representing the IPv4 address of that source node. Each sequence number field 190 holds a value (e.g., 16 bits) representing a sequence number for the corresponding source node. The Fig. 8A shows the message format for an even number of source nodes. Fig. 8B shows an alternative ending 192 for the NEW PARENT message format 186 for an odd number of source nodes.

10 CANCEL PARENT:

Fig. 9 shows an embodiment of an exemplary format 194 for a CANCEL PARENT message. The format 194 includes one or more source node identity fields 196, 196', 196'' (generally 196) for including the IPv4 address of each source node for which the CANCEL PARENT message applies.

15 HEARTBEAT:

In one embodiment, the HEARTBEAT message has an eight-bit length and holds a sequence number for the broadcast channel.

END OF BROADCAST:

In one embodiment, the END\_OF\_BROADCAST message has an eight-bit length and  
20 holds a sequence number for the broadcast channel.

Link-State Update Messages:

The TBRPF protocol provides two formats for two types of link-state update messages. One type of link-state update message, referred to as type LINK\_STATE\_UPDATE\_A, includes a single sequence number for each source node, and is therefore used only if the updates for all  
25 links coming out of the same source have the same sequence number. (For example, periodic

- 42 -

updates have this property.) This is done to reduce the message size. The second type of link-state update message, referred to as type LINK\_STATE\_UPDATE\_B, includes a separate sequence number for each link.

Fig. 10A shows an embodiment of an exemplary format 198 for one type of link-state update message, LINK-STATE\_UPDATE\_A. The format 198 includes one or more link-state updates ("lsuA") 200, 200', 200'' (generally 200). Each lsuA 200 represents an update message with respect to a particular source node and includes a source node identity field 202, 202', 202'' (generally 202), a number of neighbor nodes field 204, 204', 204'' (generally 204), and one or more neighbor node sections 206, 206', 206'' (generally 206). For each neighbor node listed in a particular lsuA 200, each neighbor node section 206 includes a neighbor-node identity field 208, 208', 208'' (generally 208), a sequence number field 210, 210', 210'' for corresponding source nodes and neighbor nodes, and a link metrics field 212, 212', 212'' (generally 212) for that neighbor node.

The source node identity field 202 holds a value (e.g., 32-bits) for the IPv4 address of the corresponding source node. The number of neighbor nodes field 204 holds a value (e.g., 16 bits) representing the number of neighbor nodes of the corresponding source node. The neighbor-node identity field 208 holds the IPv4 address of a neighbor node of the corresponding source node. The sequence number field 210 holds a value (e.g., 16 bits) representing a sequence number for the corresponding source and neighbor node. The link metrics field 212 holds a value (e.g., 32 bits) representing the link metrics associated with the neighbor node of the corresponding source node.

Fig. 10B shows an embodiment of an exemplary format 220 for the second type of link-state update message, LINK-STATE\_UPDATE\_B. The format 220 includes one or more link-state updates ("lsuB") 222, 222', 222'' (generally 222). Each lsuB 222 represents an update message with respect to a particular source node and includes a source node identity field 224,

- 43 -

224', 224'' (generally 224), a number of neighbor nodes field 226, 226', 226'' (generally 226), a sequence number field 228, 228', 228'' (generally 228), and one or more neighbor node sections 230, 230', 230'' (generally 230). For each neighbor node listed in a particular lsuB 222, each neighbor node section 230 includes a neighbor-node identity field 232, 232', 232'' (generally 232), and a link metrics field 234, 234', 234'' (generally 234) for that neighbor node.

The source node identity field 224 holds a value (e.g., 32-bits) for the IPv4 address of the corresponding source node. The number of neighbor nodes field 226 holds a value (e.g., 16 bits) representing the number of neighbor nodes of the corresponding source node. The sequence number field 228 holds a value (e.g., 16 bits) representing a sequence number for the associated with the source and neighbor nodes. The neighbor-node identity field 232 holds the IPv4 address of a neighbor node of the source node. The link metrics field 234 holds a value (e.g., 32 bits) representing the link metrics associated with the neighbor node of the corresponding source node.

#### RETRANSMISSION\_OF\_BROADCAST:

In brief, a RETRANSMISSION\_OF\_BROADCAST message provides the retransmission of a compound update message in response to a NACK message. This compound message may contain one or more atomic messages of type LINK\_STATE\_UPDATE\_A or LINK\_STATE\_UPDATE\_B concatenated together. Fig. 11 shows an embodiment of an exemplary format 240 of a RETRANSMISSION\_OF\_BROADCAST message including a message header 162''' and a compound message 180'. The message header 162''', like the message header 162 of the atomic message format 160 described above, includes a type field 166', a mode field 170', a number of sources field 172', an offset field 174', and a link sequence number field 176'. The type field 166' is set to RETRANSMISSION\_OF\_BROADCAST (e.g., = 9), and the number of sources field 172' is set to 0. The offset field 174' is the offset (in bytes) from the 0'th byte of the current compound message header to the 0'th byte of the next compound



- 44 -

message header 162' in the RETRANSMISSION\_OF\_BROADCAST message 240. A 16-bit offset value enables concatenation of compound messages 180' up to 64 kilobytes in length.

As described above, broadcast update messages can be retransmitted on unicast or broadcast channels. For retransmission on a unicast channel, the mode field 170' is set to UNICAST (e.g., =0) and the atomic message header 162''' precedes the compound message 180'. The LSEQ field 176' holds the sequence number corresponding to the unicast channel on which the message is sent. The LSEQ field 176 of each atomic message in the compound message 180' is the broadcast sequence number that was included in the original (broadcast) transmission of the message. Multiple RETRANSMISSION\_OF\_BROADCAST messages can be bundled into a compound message 180' as described above.

### Selecting a Routing Path for Transmitting Packets

Routing protocols can also be classified according to whether they find optimal (shortest) routes or sub-optimal routes. By not requiring routes to be optimal, it is possible to reduce the amount of control traffic (including routing updates) necessary to maintain the routes. However, optimal routes are desirable because they minimize delay and the amount of resources (e.g., bandwidth and power) consumed. The TBRPF protocol computes optimal routes based on the advertised link states; however, the advertised link states themselves may be approximate in order to reduce the frequency at which each link is updated.

In the full-topology embodiment of the TBRPF protocol, each routing node 14 has complete link-state information. Each routing node 14 then applies a path selection algorithm to compute preferred paths to all possible destinations, and to update these paths when link states are updated. One exemplary path selection algorithm is to apply Dijkstra's algorithm to compute shortest paths (with respect to cost, c) to all destinations. In other embodiments, the TBRPF protocol can employ any other path selection algorithm. Once preferred paths are computed, the

- 45 -

routing table entry for node  $u$  is set to the next node on the preferred path to node  $u$ . If minimum-hop routing is desired, then the routing table entry for node  $u$  can be set to the parent  $p_i(u)$ .

## 2. Neighbor Discovery

5 Each routing node 14 running the TBRPF protocol uses a neighbor discovery protocol to detect the establishment of new links to new neighbor nodes and the loss of established links to existing neighbor nodes. In general, the neighbor discovery protocol dynamically establishes bi-directional links and detects bi-directional link failures through the periodic transmission of HELLO messages. The neighbor discovery protocol is both automatic and continuous, and may  
10 include a data link-to-IPv4 address resolution capability. Because the neighbor discovery protocol is responsible for both link state maintenance and data link-to-IPv4 address resolution in the subnet 10, the neighbor discovery protocol operates as a data-link-level protocol.

Fig. 12 shows an exemplary embodiment of a process 250 used by the nodes 18 to perform neighbor discovery. The process uses the following three types of control messages:  
15 HELLO, NEIGHBOR, and NEIGHBOR ACK. This embodiment of the neighbor discovery protocol operates as follows. Every each node  $i$  in the subnet periodically transmits (step 252) a HELLO message at predetermined (e.g.,  $HELLO\_INTVL = 0.5$  seconds) timeout intervals. (The  $HELLO\_INTVL$  value is common to all nodes 18 within the subnet 10, but different subnets may use different  $HELLO\_INTVL$  values.) HELLO messages are sent to the data link level  
20 broadcast address and includes the identity of transmitting node  $i$ .

A node  $j$  receiving a HELLO message from a new neighbor, node  $i$ , responds (step 254) with a NEIGHBOR message containing the identity of node  $j$ , sending the NEIGHBOR message to the data link unicast address of the new neighbor node  $i$ . Then, upon receiving the NEIGHBOR message, node  $i$  sends (step 256) a NEIGHBOR ACK to node  $j$  using the data link  
25 unicast address of node  $j$ . The NEIGHBOR ACK message contains the identity of node  $i$ . The

NEIGHBOR and NEIGHBOR ACK messages also contain the current link-level sequence number for the broadcast channel (discussed below). Thus, a link from node i to node j is established by node i receiving a NEIGHBOR packet from node j, and a link from node j to node i is established by node j receiving a NEIGHBOR ACK packet from node i. The link to an  
5 existing neighbor is declared to be down if no traffic (including HELLO messages and ACKs) has been received from the neighbor node within a predetermined time interval (e.g., within the last LINKDOWN\_INTVL = 2.0 seconds).

Implementations of this embodiment of the neighbor discovery protocol should detect the event of a data link-to-IP address mapping change for existing links. This may occur in one of  
10 the following instances:

1. Two or more nodes in the subnet 10 are using the same IP address.
  2. An existing node in the subnet 10 has changed its data link layer address.
  3. A new node is now using the IP address of a former node that may have left the subnet 10.
- 15 In the first case, the implementation should print some form of "duplicate IP address detected" message to the console. In the second and third instances, the cached link state should be updated to reflect the new data link-to-IPv4 address mapping.

Fig. 13 shows an exemplary embodiment of a packet format 260 for the HELLO, NEIGHBOR, and NEIGHBOR ACK neighbor discovery protocol messages on the subnet 10.  
20 The data link header for each message is not shown, since it is specific to the underlying data link layer.

The eight-bit "Type" field 262 indicates the type of message. For example, each message can be identified by the following examples of values in the Type field 262:

25	HELLO	10
	NEIGHBOR	11

- 47 -

NEIGHBOR\_ACK 12

The eight-bit "BCAST Seq# field 264 indicates a sequence number from 0...15 (4 bits), used in NEIGHBOR and NEIGHBOR ACK messages as described above. The four address fields (sender hardware address 266; sender protocol address 268; target hardware address 270; target protocol address 272) facilitate the address resolution process. The fields 266, 268, 270, and 272 contain the following examples of values, based on the type of neighbor discovery message:

Message type is HELLO

Sender Hardware Address 266: data link address of sender  
 Sender Protocol Address 268: IPv4 address of sender  
 Target Hardware Address 270: data link broadcast address  
 Target Protocol Address 272: unused

Message type is NEIGHBOR

Sender Hardware Address 266: data link address of sender  
 Sender Protocol Address 268: IPv4 address of sender  
 Target Hardware Address 270: sender H/W Address from received HELLO  
 Target Protocol Address 272: sender IP Address from received HELLO

Message type is NEIGHBOR ACK

Sender Hardware Address 266: data link address of sender  
 Sender Protocol Address 268: IP address of sender  
 Target Hardware Address 270: sender H/W address from NEIGHBOR  
 Target Protocol Address 272: sender IP address from NEIGHBOR

Usage of the other fields 274, 276, 278, and 280 in the packet 260 are described "An Ethernet Address Resolution Protocol: Or Converting Network Protocol Addresses To 48.Bit Ethernet Addresses For Transmission On Ethernet Hardware," by David C. Plummer, Request for Comments (RFC) No. 826, November 1982.

- 48 -

### Reduced Overhead Hello Protocol

Another embodiment of the neighbor discovery protocol, hereafter referred to as Reduced Overhead Hello Protocol (ROHP), is suited for MANETs. As described further below, the ROHP is suited for MANETs because the protocol can operate correctly although an asymmetric (unidirectional) link may exist between any two nodes at any time, link states may change frequently due to node mobility and interference, and the channel may be noisy so that not all transmitted packets are successfully received by all neighbor nodes. An objective of ROHP is to allow each node 18 in the MANET 10 to quickly detect the neighbor nodes with which that node 18 has a direct and symmetric link, (i.e., a bi-directional link such that the node at each end of the link can hear the other node.) The ROHP also detects when a symmetric link to some neighbor no longer exists.

In brief overview, the ROHP reports each change in the state of a neighbor node (e.g., “heard”, “symmetric”, or “lost”) in only the next K HELLO messages, where K is a small positive integer (e.g., K = 3 to 5) such that a node declares a neighbor to be “lost” if it does not receive any complete HELLO message from that neighbor node within a time period equal to K number of time intervals. Each time interval is hereafter referred to as HELLO\_INTERVAL, which is for example 0.5 to 2 seconds. In contrast, each HELLO message of conventional neighbor discovery protocols (e.g., as in OSPF and OLSR (Optimized Link-State Routing Protocol)) includes the identities (or addresses) of all neighbors.

### Neighbor Table

Each node 18 maintains a neighbor table, which has an entry for each known neighbor node and stores state information for that neighbor node. An entry for neighbor node B, for example, contains the following variables:

state(B): The current state of the link to neighbor node B, which can be “heard”, “symmetric”, or “lost”.

- 49 -

hold\_time(B): The amount of time (in seconds) remaining before state(B) must be changed to "lost" if no further complete HELLO message from B is received.

counter(B): The number of subsequent HELLO messages that include the identity of the neighbor node B in the list corresponding to state(B).

5       The entry for neighbor node B may be deleted from the table if state(B) remains equal to "lost" for a period not less than  $K * \text{HELLO\_INTERVAL}$ .

Three possible states of a neighbor node B have the following meaning at node A:

10       "Heard":       A complete HELLO message was received from neighbor node B within the last  $K * \text{HELLO\_INTERVAL}$  seconds, but it is unknown whether neighbor node B can hear node A.

      "Symmetric": Nodes A and B can hear each other.

      "Lost":       No complete HELLO message has been received from neighbor node B within the last  $K * \text{HELLO\_INTERVAL}$  seconds.

#### **Sending HELLO Messages**

15       Each node 18 sends a HELLO message periodically every  $\text{HELLO\_INTERVAL}$  seconds, possibly with a small jitter to avoid repeated collisions. Because of message size limitations that may be imposed by the MANET 10, a HELLO message may be too large to send within one packet, in which case, the sending node 18 sends the HELLO message in multiple packets within a period equal to the  $\text{HELLO\_INTERVAL}$ . Depending on the implementation of the ROHP, the  
20       receiving node may or may not be able to extract information from a partially received HELLO message.

A HELLO packet sent by a node includes the following information:

1).     The identity (e.g., IP address) of the sending node.



- 50 -

2). A list of all neighbor nodes that recently changed to the “heard” state. More specifically, a list of identities of neighbor nodes B such that  $\text{state}(B) = \text{“heard”}$  and  $\text{counter}(B) > 0$ .

3). A list of all neighbor nodes that recently changed to the “symmetric” state. More specifically, a list of identities of neighbor nodes B such that  $\text{state}(B) = \text{“symmetric”}$  and  $\text{counter}(B) > 0$ .

4). A list of all neighbor nodes that recently changed to the “lost” state. More specifically, a list of identities of neighbor nodes B such that  $\text{state}(B) = \text{“lost”}$  and  $\text{counter}(B) > 0$ .

10 Whenever a neighbor node B is included in one of the above three lists,  $\text{counter}(B)$  decrements by 1. As a result, each state change is included in at most K HELLO messages, and in some cases (as described below) is not included in any HELLO message. HELLO messages can also contain other information, as discussed below.

### Receiving a HELLO Message

15 Fig. 14 shows an exemplary embodiment of a process by which each node 18 operating according to the ROHP neighbor discovery processes a received HELLO message. In step 288, a node (referred to as receiving node A) receives a partial or complete HELLO message.

Because a HELLO message must be transmitted within a time interval of length HELLO\_INTERVAL, the receiving node A declares the HELLO message to be partial if not all  
20 of its parts have been received within a time interval of this length. If the HELLO message is complete and an entry for neighbor node B does not exist in the table, the receiving node A creates (step 290) such an entry with  $\text{state}(B) = \text{“lost”}$ . If the HELLO message is complete, receiving node A also sets (step 292) the variable  $\text{hold\_time}(B)$  to  $K * \text{HELLO\_INTERVAL}$ .

The value of the variable  $\text{hold\_time}(B)$  decreases to 0 (expires) if no HELLO message from  
25 neighbor node B is subsequently received within  $K * \text{HELLO\_INTERVAL}$  seconds. When

- 51 -

hold\_time(B) expires, the receiving node A sets state(B) to "lost" and counter(B) to K. This indicates that the receiving node A is to include the identity of node B in the list of "lost" neighbor nodes in the transmission of the next K HELLO messages or until state(B) changes again (whichever occurs first).

- 5 The receiving node A then performs (step 294) an action based on whether the received HELLO message is complete, whether the receiving node A appears in a list within the received HELLO message, and if so, which list, and the current state of the neighbor node B (i.e., state(B)). The actions performed by the receiving node A are summarized in Table 3 below.

TABLE 3

state(B)	Action			
	<u>Receiving node A is in not in any list</u>	<u>Receiving node A is in "lost" list</u>	<u>Receiving node A is in "heard" list</u>	<u>Receiving node A is in "symmetric" list</u>
lost	If msg is complete, set state(B) to "heard" and counter(B) to K	If msg is complete, set state(B) to "heard" and counter(B) to K	If msg is complete, set state(B) to "symmetric" and counter(B) to K	If msg is complete, set state(B) to "symmetric" and counter(B) to K
heard	No action	No action	Set state(B) to "symmetric" and counter(B) to K	Set state(B) to "symmetric" and counter(B) to 0
symmetric	No action	Set state(B) to "heard" and counter(B) to 0	If counter(B) = 0, set counter(B) to K	Set counter(B) to 0

- 10 Accordingly, upon receiving a complete or partial HELLO message from neighbor node B, the action performed by the receiving node A is as follows.

1. If state(B) = "lost" and the HELLO message is complete, and if the message does not include node A in any list or includes node A in the "lost" list, then set state(B) to "heard" and counter(B) to K. If state(B) = "lost" and the HELLO message is complete,

- 52 -

and if the message includes node A in the “heard” or “symmetric” list, then set state(B) to “symmetric” and counter(B) to K.

2. If state(B) = “heard” and the message includes node A in the “heard” list, then set state(B) to “symmetric” and counter(B) to K. If state(B) = “heard” and the message includes node A in the “symmetric” list, then set state(B) to “symmetric” and counter(B) to 0. (In this case, the receiving node A need not include node B in its HELLO messages, since both nodes A, B already know that the link is symmetric.)

3. If state(B) = “symmetric” and the message includes node A in the “heard” list and counter(B) = 0, then set counter(B) to K. If counter(B) > 0, then counter(B) need not be set to K, because the “heard” entry is just a repeat of one that was included in a recently received HELLO message from B. If state(B) = “symmetric” and the message includes node A in the “symmetric” list, then set counter(B) to 0. (Both nodes know that the link is symmetric.) If state(B) = “symmetric” and the message includes receiving node A in the “lost” list, then set state(B) to “heard” and counter(B) to 0. (node B cannot hear node A, but node A can hear node B.)

Note that a complete HELLO message must be received in order to create a new entry in the neighbor node table or to change the state of a neighbor node from “lost” to “heard” or to “symmetric.” This prevents the creation of a link that has poor quality.

### Variations of ROHP

20 In other embodiments, HELLO messages can be augmented to include enough information to inform each neighbor node of the set of neighbor nodes with which the sending node has symmetric links. This can be accomplished by setting the counter(B) to K, rather than to 0 (see case 2 above), so that node B is included in the “symmetric” list of the next K HELLO messages, even though nodes A and B already know that the link between them is symmetric.

- 53 -

In addition, node A can inform any new neighbor node of the set of neighbor nodes with which node A has symmetric links. Node A can distribute this information by (a) including the set of all neighbor nodes to which symmetric links exist in the next K HELLO messages, whenever the state of the new neighbor node changes to "symmetric"; or (b) sending this information in a separate message that is unicast reliably to the new neighbor node.

HELLO messages can also be augmented to include other information, such as link metrics, sequence numbers, states of non-adjacent links, time stamps, designated routers, special relays, and other data.

The ROHP can be used in conjunction with any routing protocol that uses HELLO messages for neighbor discovery, such as TBRPF (described herein), OSPF, and OLSR. An advantage of the ROHP over existing neighbor discovery protocols, such as those discovery protocols used within OSPF and OLSR, is that ROHP employs HELLO messages that on average are smaller than such neighbor discovery protocols because each neighbor state change observed by a node is included in at most K HELLO messages (unlike OSPF and OLSR), resulting in reduced communications overhead and bandwidth consumption. In addition, because HELLO messages are smaller, they can be sent more frequently, resulting in a faster detection of topology changes.

### **3. IPv6-IPv4 Compatibility Address Format**

Referring back to the subnet 10 in Fig. 1, assume, for example, that the nodes 18 of subnet 10 belong to one domain, and that the gateway 16 is the border gateway 16 for that domain. Assume also that both the IP host A 12 and the gateway 16 are IPv6 nodes 18, and that the other nodes 18 in the subnet 10 are IPv4 nodes, without any IPv6 routing capability. Any route taken by packets sent by the IP host A 12 to the server 40 on the Internet 30 necessarily traverses IPv4 infrastructure to reach the gateway 16. To communicate across the subnet 10 with the heterogeneous IP infrastructure, the IP host 12 and the gateway 16 use an aggregatable,

- 54 -

global, unicast addresses, hereafter referred to as an “IPv6-IPv4 compatibility address.” The use of IPv6-IPv4 compatibility addresses enables IPv6 nodes (1) to forward IPv6 packets across native IPv6 routing infrastructure or (2) to automatically tunnel IPv6 packets over IPv4 routing infrastructure without requiring a pre-configured tunnel state. A routing node 14 with an IPv6-IPv4 compatibility address can serve as a router for nodes 18 with native IPv6 addresses (i.e., IPv6 addresses that are not IPv6-IPv4 compatibility addresses) connected to the same link. On behalf of such native IPv6 nodes, the IPv6-IPv4 routing node 14 can automatically tunnel messages across the IPv4 infrastructure of the subnet 10 to reach the border gateway 16.

Fig. 15A shows an exemplary embodiment of a format 300 for IPv6-IPv4 compatibility addresses. The format 300 includes a 64-bit address prefix 302 and a 64-bit interface identifier 304. The address prefix 302 specifies a standard 64-bit IPv6 routing prefix, such as that described in the Internet RFC (request for comment) #2374. The address prefix 302 includes a 3-bit Format Prefix (FP) 303, which for all IPv6-IPv4 compatibility addresses is set to “001”, and aggregation identifiers 305. Consequently, the format prefix 303 and the topologically correct aggregation identifiers 305 of the IPv6-IPv4 compatibility addresses are the same as those of IPv6 addresses assigned to IPv6 nodes, enabling IPv6 nodes to route IPv6 packets using IPv6-IPv4 compatibility addresses across IPv6 infrastructure. The 64-bit interface identifier 304 is a specially constructed 64-bit global identifier interface identifier (i.e., 64-bit EUI-64).

Fig. 15B shows an embodiment of the interface identifier 304 including a 24-bit company identifier 306 concatenated with a 40-bit extension identifier 308. In one embodiment, the 24-bit company identifier 306 is a special IEEE Organizationally Unique Identifier (OUI) reserved by the Internet Assigned Numbers Authority (IANA) for supporting the IPv6-IPv4 compatibility addresses. The IEEE Registration Authority (IEEE/RAC) assigns the OUI to an organization and the organization owning that OUI typically assigns the 40-bit extension identifier 308. In Fig. 15B, the string of ‘c’s represents the company-specific bits of the OUI, the bit ‘u’ represents

the universal/local bit, the bit 'g' represents the individual/group bit and the string of 'm's are the extension identifier bits. Here, when the bit 'u' equals 1, the scope of the address is global and when the bit 'u' equals 0, the scope is local.

To support encapsulation of legacy IEEE EUI-48 (24-bit) extension identifier values, the first two octets of the 40-bit extension identifier 308 (i.e., bits 24 through 39 of the address) are set to 0xFFFE if the extension identifier 308 encapsulates an EUI-48 value. Further, the first two octets of the extension identifier 308 are not set to 0xFFFF, as this value is reserved by the IEEE/RAC. All other 40-bit extension identifier values are available for assignment by the addressing authority responsible for a given OUI. Thus, as described further below, the IPv6-IPv4 compatibility address format 300 enables embedding an IPv4 address in an IPv6-IPv4 compatibility address without sacrificing compliance with the EUI-64 bit format.

Fig. 15C shows an embodiment of the interface identifier 304 including an OUI field 306, a type field 310, a type-specific extension field (TSE) 312, a type-specific data field (TSD) 314. As shown, the OUI field 306 includes the OUI of IANA, (e.g., 00-00-5E), with 'u' and 'g' bits. The type field 310 indicates how the TSE 312 and TSD 314 fields are interpreted; in general, the type field 310 indicates whether the interface identifier 304 encapsulates an IPv4 address that is suitable for automatic intra-subnet IPv6-in-IPv4 tunneling. Table 1 shows the interpretations of TSE and TSD for various values in the type field 310:

TABLE 1

<u>TYPE Value</u>	<u>(TSE, TSD) Interpretation</u>
0x00-0xFD	RESERVED
0xFE	(TSE, TSD) together contain an embedded IPv4 address
0xFF:	TSD is interpreted based on the value of TSE as shown in TABLE 2

TABLE 2

<u>TSE Value</u>	<u>TSD Interpretation</u>
0x00-0xFD	RESERVED for future use (e.g., by IANA)

0xFE	TSD contains 24-bit EUI-48 interface identifier
0xFF	RESERVED (e.g., by IEEE/RAC)

Thus, if an IPv6-IPv4 compatibility address has TYPE=0xFE, the TSE field 312 is treated as an extension of the TSD field 314, which indicates that the IPv6-IPv4 compatibility address includes a valid IPv6 prefix and an embedded IPv4 address.

If the IPv6-IPv4 compatibility address has TYPE=0xFF, the TSE field 312 is treated as an extension of the TYPE field 310. When TSE = 0xFE, the TSD field 314 includes a 240bit EUI-48 interface identifier. Thus, the IPv6-IPv4 compatibility address format 300 conforms to all requirements of a 64-bit global identifier (i.e., the EUI-64 format) and supports encapsulation of EUI-48 interface identifiers (i.e., when TSE = 0xFE). For example, an existing IANA EUI-48 format multicast address such as:

01-00-5E-01-02-03

is written in the IANA EUI-64 format as:

01-00-5E-FF-FE-01-02-03.

Other values for TYPE and, hence, other interpretations of the TSE and TSD fields 312, 314 are reserved for future use.

Fig. 15D shows a specific example of an IPv6-IPv4 compatibility address 316 for a node 18 with an IPv4 address of 140.173.189.8. This IPv4 address may be assigned an IPv6 64-bit address prefix 302 of 3FFE:1a05:510:200::/64. Accordingly, the IPv6-IPv4 compatibility address 316 for this IPv4 node is expressed as:

3FFE:1a05:510:200:0200:5EFE:8CAD:8108

In an alternative form, the IPv6-IPv4 compatibility address 316 with the embedded IPv4 address is expressed as:

3FFE:1a05:510:200:0200:5EFE:140.173.189.8



- 57 -

Here, the least significant octet of the OUI (02-00-5E) in the interface identifier 304 is 0x02 instead of 0x00 because the bit 'u' is set to 1 for global scope.

Similarly, the IPv6-IPv4 compatibility addresses for the link-local and site-local (i.e., within the subnet 10) variants, respectively, of are:

5        FE80::0200:5EFE:140.173.189.8  
      FEC0::200:0200:5EFE:140.173.189.8

As previously noted, the IPv6-IPv4 compatibility address format 300 enables IPv6 nodes to tunnel IPv6 packets through a one-time IPv6-in-IPv4 tunnel across IPv4 routing infrastructure. Fig. 15E shows an embodiment of a packet header 320 used for tunneling IPv6 packets using

10    IPv6-IPv4 compatibility addresses across IPv4 routing infrastructure. In this embodiment, the header 320 includes a 20-byte IPv4 header 322 and a 40-byte IPv6 header 324. The IPv6 header 324 includes an IPv6 address 329 of the node that is the source of the IPv6 packet and an IPv6-IPv4 compatibility address 316 associated with the final IPv6 destination node. The IPv4 header 322 includes the IPv4 address 326 of the dual-stack node that "re-routes" the IPv6 packet by

15    tunneling the IPv6 packet through the IPv4 routing infrastructure. The IPv4 header 322 also includes the IPv4 address 328 of an IPv4 destination node that typically is the same as the IPv4 address embedded within the IPv6 destination address' IPv6-IPv4 compatible interface identifier 304. Alternatively, the IPv4 address 328 can be the IPv4 address 328 of the next-hop IPv6 gateway that has a path to the final IPv6 destination address and, therefore, can forward the IPv6

20    packet towards the final IPv6 destination node.

Upon receiving the tunneled IPv6 packet, the IPv4 destination node determines that the IPv6 header 324 includes an IPv6-IPv4 compatibility address 316 and can route the IPv6 packet to the IPv6 destination node identified by that IPv6-IPv4 compatibility address.

- 58 -

**Address Aggregation**

One advantage of embedding an IPv4 address in the interface identifier 304 of an IPv6 address is that large numbers of IPv6-IPv4 compatibility addresses 316 can be assigned within a common IPv6 routing prefix 302, thus providing aggregation at the border gateway 16. For example, a single 64-bit IPv6 prefix 302 for the subnet 10, such as 3FFE:1a05:510:2418::/64, can include millions of nodes 18 with unique IPv4 addresses embedded in the interface identifier 304 of the IPv6-IPv4 compatibility addresses. This aggregation feature allows a “sparse mode” deployment of IPv6 nodes throughout a large Intranet comprised predominantly of IPv4 nodes.

**Globally and Non-Globally Unique IPv4 addresses**

Another advantage is that IPv6-IPv4 compatibility addresses 316 support subnets that use globally unique IPv4 address assignments and subnets that use non-globally unique IPv4 addresses, such as when private address assignments and/or network address translation (NAT) are used.

**Non-globally Unique IPv4 addresses**

IPv4 addresses need not be globally unique but may be allocated through a private network-addressing scheme that has meaning only within the context of that domain. IPv6-IPv4 compatibility addresses for private IPv4 addresses set the ‘u’ bit to 0 for local scope. For example, a node with the private, non-globally unique IPv4 address 10.0.0.1 can be assigned the IPv6-IPv4 compatibility address of

3FFE:1a05:510:200:0000:5EFE:10.0.0.1,  
which uses the same example IPv6 64-bit prefix and IANA OUI (00-00-5E) described above with the ‘u’ bit in the EUI-64 interface identifier indicating that this is a local address

- 59 -

### **Routing with IPv6-IPv4 Compatibility Addresses**

By embedding an IPv4 address in the interface identifier 304 of an IPv6-IPv4 compatibility address 316, IPv6 packets can be routed globally over the IPv6 infrastructure or tunneled locally across portions of the IPv4 infrastructure of the subnet 10 that have no IPv6 routing support. Thus, the compatibility-addressing scheme supports heterogeneous IPv6/IPv4 infrastructures in transition with incremental deployment of IPv6 nodes within the subnet 10.

### **Intra-domain Routing**

Fig. 16 shows an exemplary embodiment of an intra-domain routing process 330 by which a routing node 14, configured with IPv6 and IPv4 routing tables, routes a packet having the IPv6-IPv4 compatibility address. Upon receiving the packet, the routing node 14 has IPv6 node software that checks (step 332) for the special IETF OUI 306 and the type field 310 encapsulated in the interface identifier 304. If the software finds the special OUI 306 and the value of 0xFE in the type field 310, this means that the received packet has an IPv6 prefix and an embedded IPv4 address.

The routing node 14 then determines (step 334) if any IPv6 routing information leads to the destination node 18; that is, if the IPv6 routing table has an entry for 'default' (i.e., the default gateway) or for the IPv6 prefix of the destination node 18. If such an entry is found, the router 14 determines (step 336) whether there is a path through IPv6 routing infrastructure to the gateway 16 for the IPv6 prefix of the destination node 18. If there is such an IPv6 path, then the router 14 sends (step 338) the packet as an IPv6 packet to the IPv6 gateway 16 for that IPv6 prefix.

If no such IPv6 path to the gateway 16 through IPv6 routing infrastructure exists, the routing node 14 construes (step 340) the last four bytes of the extension identifier 308 as an IPv4 address embedded in the IPv6-IPv4 compatibility address. The routing node 14 then determines

- 60 -

(step 342) if the IPv4 routing table includes an entry for a prefix of the embedded IPv4 address of the destination.

Upon finding such an entry, the routing node 14 encapsulates (step 342) the IPv6 packet for tunneling through the IPv4 routing infrastructure using the embedded IPv4 address as the destination for the tunneled packet. (The general format for an encapsulated packet is shown in Fig. 15E.) One technique for automatically tunneling the IPv6 packet is described in "Transition Mechanism for IPv6 Hosts and Routers," by R. Gilligan and E. Nordmark, draft-ietf-ngtrans-mech-04.txt (work in progress). This technique can also be applied to the IPv6-IPv4 compatibility address. This implies that the gateway 16 also uses IPv6-IPv4 compatibility addresses.

#### **Inter-domain Routing:**

##### **Globally Unique IPv4 Addresses without Privacy Concerns**

Where nodes 18 within an heterogeneous IPv6/IPv4 subnet 10 use globally unique IPv4 addresses and where no privacy concerns exist regarding exposure of internal IPv4 addresses to the public Internet, messages may be routed across domain boundaries using the same routing process 330 described above in Fig. 16.

##### **Globally Unique IPv4 Addresses without Privacy Concerns**

One advantage of the IPv6-IPv4 compatibility address format 300 is that the format 300 does not necessarily expose the true identification of the sending node, if an administrative authority for the subnet 10 wishes to enforce a policy of not exposing internal IPv4 addresses outside of the subnet 10. To accomplish this, the administrative authority configures the border gateway 16 of the subnet 10 to perform a type of "reverse network address translation," which transforms the IPv6-IPv4 compatibility address interface identifier 304 with embedded IPv4 address of the sending node into an anonymous ID for inter-domain routing outside the subnet 10. Within the subnet 10, the fully qualified IPv6-IPv4 compatibility address interface identifier

- 61 -

304 with the embedded IPv4 address of the sending node is still used to enable automatic IPv6-in-IPv4 tunneling, and the intra-domain routing of IPv6 packets follows the process 330 described above.

In one embodiment, the border gateway 16 advertises an IPv6 prefix 302 of 2002::/16 and the IPv6 prefix 302 of 2002:V4ADDR/48 where 'V4ADDR' is the globally unique embedded IPv4 address of the border gateway 16. IPv6-IPv4 compatibility addresses within the subnet 10 are constructed as the concatenation of a 2002:V4ADDR/48 prefix, a 16-bit SLA ID, and a 64-bit EUI64 interface identifier 304 as described above.

For example, if the IPv4 address of the border gateway is 140.173.0.1, the IPv4 address of the IPv4 node within the subnet 10 is 140.173.129.8 and the node resides within SLA ID 0x001, the IPv6-IPv4 compatibility address 316 within the subnet is constructed as:

2002:8CAD:1:1:0200:5EFE:8CAD:8108,

where the '2002:' is a predetermined prefix associated with the reverse network address translation; the '8CAD:1:' is the IPv4 address (140.173.0.1) of the border gateway 16; the second '1:' is the SLA ID; the '0200:5EFE' is the IANA-specific OUI (with the 'u' bit set to global scope and the type field 310 indicating that the compatibility address includes an embedded IPv4 address; and the '8CAD:8108' is the embedded IPv4 address (140.173.129.8) of the internal IPv4 node.

The border gateway 16 performs "reverse network address translation" using an identifier not vulnerable to eavesdropping. The border gateway 16 maintains a mapping of the identifier to the actual IPv4 address of the IPv4 node in order to map messages from destinations back to the actual IPv4 node within the subnet 10. For example, if the border gateway 16 replaced the IPv4 address 140.173.129.8 with the identifier value: 0x00000001, the IPv6-IPv4 compatibility address outside the subnet 10 is constructed as:

2002:8CAD:1:1:0000:5EFE:0:1

- 62 -

Here: again the least significant octet of the EUI-64 interface identifier 304 has the 'u' bit set to 0 to indicate that the embedded IPv4 address is not globally unique.

The IPv6-in-IPv4 tunneling for inter-domain routing then derives the IPv4 source address from the IPv4 address of the numerous separate tunnel transitions for an IPv6 packet traveling  
5 from a sending node to a destination node. The transitions include (1) intra-domain tunnels from the IPv6 sending node through routers along the path to the border gateway for its domain, (2) inter-domain tunnels from the sending node's border gateway through other transit routers along the path to a border gateway for the destination, and (3) intra-domain tunnels from the destination node's border gateway through intra-domain routers along the path to the destination  
10 node itself. Thus, IPv4 addresses within the subnet are exposed across the public Internet 30.

#### **Non globally unique IPv4 addresses**

Embodiments of the subnet 10 that use private, non-globally unique IPv4 addresses require a border gateway 16 that implements an inter-domain routing function as described above. For example, if the IPv4 address of the border gateway 16 is 140.173.0.1, the IPv4  
15 address of an IPv4 node within the subnet 10 is 10.0.0.1, and the IPv4 node resides within SLA ID 0x001, the IPv6-IPv4 compatibility address within the subnet 10 is constructed as:

2002:8CAD:1:1:0000:5EFE:0A00:1,

where again the least significant octet of the EUI-64 interface identifier 304 has the 'u' bit set to 0 to indicate that the embedded IPv4 address '0A00:1' (10.0.0.1) is not globally unique.

20 The administrative authority for such embodiments of the subnet 10 may institute a policy that permits exposing non-globally unique IPv4 addresses to the public Internet 30. In this case, the reverse network address translation is unnecessary, but might be used to protect against eavesdropping on the non-globally unique addresses.

- 63 -

**Additional Routing Considerations**

In a different embodiment than that described in Fig. 16, each host 12 or router 14 that sends an IPv6 packet to an IPv6-IPv4 compatibility destination address follows the following process:

- 5        If the 64-bit IPv6 prefix of the IPv6-IPv4 compatibility destination address matches the 64-bit IPv6 prefix of one of the network interfaces, tunnel the packet through IPv4. Otherwise, route the packet through IPv6.

From the above sending process, a sending node that does not have an interface which shares a common 64-bit routing prefix with the packet's IPv6-IPv4 compatibility destination address sends the packet to the next-hop gateway determined by an IPv6 routing table lookup. In short, when a sending node does not have an interface which shares a common 64-bit (site-level) routing prefix with an IPv6-IPv4 compatibility destination address, the sending rule is identical to that for a native IPv6 destination address. This decision is independent of whether the sending node has an IPv6-IPv4 compatibility address itself, or whether the sending node even comprises a dual-stack configuration. The sending node can be a native IPv6 node with no legacy IPv4 support.

When a sending node has an interface which shares a common 64-bit routing prefix with an IPv6-IPv4 compatibility destination address, the sending node must assume that the destination is not directly reachable at the data-link level, although the shared site-level routing prefix implies otherwise. Instead, if the sending node comprises a dual-stack configuration, it automatically tunnels the IPv6 packet to the IPv4 address embedded within the IPv6-IPv4 compatibility destination address' interface identifier. If the sending node is an IPv6-only node that does not comprise a dual-stack configuration, however, it has no means for automatically tunneling the packet via IPv4. In this case:



- 64 -

If the sending node is the host that originates the packet, the sending node sends the packet to a router that lists the 64-bit prefix in its router advertisements. If no such router exists, the sending node should drop the packet and return a "No route to host" error indication to the originating application. If the sending node is a router that forwards the  
5 packet, the sending node drops the packet and sends an ICMPv6 "Destination Unreachable" message to the source

By implication, the scheme breaks down if a packet with an IPv6-IPv4 compatibility destination address reaches an IPv6-only router that has an interface that shares a common 64-bit routing prefix with the IPv6-IPv4 compatibility destination address. Additional mechanisms to address  
10 this issue may be possible, such as allowing dual-stack routers to advertise 96-bit prefixes which incorporate the special 32-bit EUI-64 interface identifier prefix: 0200:5EFE. A sending node can then interpret such an advertisement to mean that the advertising router comprises a dual stack and is capable of intra-site IPv6-in-IPv4 tunneling.

#### **Incremental IPv6 Deployment Examples**

15 When deploying an IPv6 node in a subnet that is predominantly IPv4, the embedded IPv4 address within an IPv6-IPv4 compatibility assigned to that IPv6 node does not need to be globally unique. The embedded IPv4 address needs only be topologically correct for and unique within the context of that subnet 10. Also, when deployed in a predominantly IPv4 subnet, the deployed IPv6 node is unlikely to share a common multiple access data-link with an IPv6 router  
20 14 in the subnet. Because the IPv6 node does not share a common multiple access data-link with the IPv6 router, no router advertisements are available. IPv6-IPv4 compatibility addresses enable the IPv6 node to join the global IPv6 network (i.e., on the Internet 30) by automatically tunneling IPv6 messages through the intra-site IPv4 routing infrastructure. For this purpose, the deployed IPv6 node requires two pieces of static configuration information: the 64-bit IPv6  
25 network prefix for the subnet 10 and the IPv4 address of the dual-stack IPv6 gateway 16 servicing the subnet 10. No other pre-configured tunnel state information is required.

- 65 -

For example, consider a researcher who wishes to configure IPv6 on his existing IPv4-based workstation, but the network administrators for the subnet 10 have not yet configured an IPv6 router for the workstation's LAN. The researcher is aware of a dual-stack IPv6 router elsewhere within the subnet 10 (which may be several IPv4 router hops away from his workstation's LAN) and sets the 64-bit IPv6 address prefix and IPv4 address of the router as configuration information on his workstation.

This configuration information is used to construct two IPv6-IPv4 compatibility addresses. One is the concatenation of the IPv6 prefix and the IPv4 address of the router to construct the IPv6-IPv4 compatibility address for the router. The researcher's workstation uses this IPv6-IPv4 compatibility address of the router as its default IPv6 gateway address. The second address is the concatenation of the IPv6 prefix and the IPv4 address of the researcher's workstation to construct the IPv6-IPv4 compatibility address which the workstation uses as its own IPv6 source address. The researcher's workstation can now access the global IPv6 Internet by first tunneling messages through the subnet-local IPv4 routing infrastructure to the IPv6 router. The IPv6 router then routes the IPv6 messages. No static configuration information is needed on the IPv6 router on behalf of the researcher's workstation.

As another example, a network administrative authority wishes to configure IPv6 on an existing IPv4 subnet under their jurisdiction, but the subnet is separated from the IPv6 border gateway 16 for the subnet by other IPv4 subnets, which are not ready for IPv6 deployment. The administrator configures a dual-stack IPv6 router (or routers) for his administrative domain by arranging for SLA (site-level aggregation)-based subnet allocation(s) from the owner of the IPv6 border gateway for the subnet. The administrator further sets the 64-bit IPv6 address prefix and IPv4 address of the border gateway as configuration information on his router. The router(s) for the administrative domain can now access the global IPv6 Internet by first tunneling messages through the site-local IPv4 routing domain to the IPv6 border gateway for the site. Hosts and/or

- 66 -

other IPv6 routers which share a common multiple access data-link with the router receive router advertisements from which they can construct native IPv6 addresses with topologically-correct 64-bit prefixes and interface identifiers via address auto-configuration. The IPv6 border gateway for the site need only have routing information that points to the router(s) for the SLA-based  
5 subnet allocations.

### **Automatic Deprecation**

As seen in the above deployment examples, the IPv6-IPv4 compatibility address format enables incremental IPv6 deployment for hosts and routers within sites that have incomplete or “sparse” IPv6 coverage at the network infrastructure level. In general, IPv6-IPv4 compatibility  
10 addresses are intended for use by nodes 18 that do not receive router advertisements because such nodes 18 do not share a common multiple access data-link with an IPv6 router. When router advertisements become available, such as when an IPv6 router is deployed on a common multiple access data-link shared by the node 18, the node 18 can discontinue use of its IPv6-IPv4 compatibility address and adopt an IPv4 unicast address using address auto-configuration for a  
15 prefix discovered through router discovery. In this way, IPv6-IPv4 compatibility addresses can gradually and automatically disappear as IPv6 nodes become widely deployed within the subnet 10. The following automatic deprecation rule for hosts and routers using IPv6-IPv4 compatible addresses can be used to transition from the use of IPv6-IPv4 compatibility addresses:

While no IPv6 router advertisements are received, continue to use the IPv6-IPv4  
20 compatibility address. If router advertisements ensue, discontinue use of the IPv6-IPv4 compatibility address and construct a native IPv6 address based on prefix information carried in the router advertisements.

### **Address Selection**

To ensure efficient routing within the destination’s subnet when multiple IPv6 destination  
25 addresses alternatives are available, a “second-tier” address selection policy is used for choosing between an IPv6-IPv4 compatibility addresses and a native IPv6 address. If multiple alternatives

- 67 -

remain after address selection has been applied on the 64-bit routing prefixes, and if at least one of the remaining alternatives is constructed with a native IPv6 interface identifier (one that does not contain an embedded IPv4 address), select a native IPv6 address. Otherwise, select an IPv6-IPv4 compatible address.

5     **4.     Updating Information upon Resuming Interrupted Communications**

Referring again to Fig. 1, assume that the mobile node 12, hereafter "client 12", and the server 40 are communicating over a route or path through the subnet 10 that includes one or more wireless links. Movement by the client 12 or by another node 14 in the subnet 10 may cause the client 12 to move in and out of communication range of the subnet 10. For example, 10 the client 12 may move to a new position in the subnet 10 (as indicated by arrow 27) or to the foreign subnet 20 (as indicated by arrow 29). While moving, the client 12 may break current a communication link (e.g., link 24) to the subnet 10 and be out of range of all routing nodes 14 within the subnet 10. As another example, the node B may move out of range of the client 12, placing the client 12 out of range of the subnet 10 if the client 12 is not within range of another 15 routing node 14 in the subnet 10. Consequently, the client 12 is not communicating with the server 40 and may not access information, particularly updated information, from the server 40. The inability to obtain updated, timely information may cause resources associated with the client 12 to be inefficiently used and adversely affect the operation of the client 12.

To lessen any adverse impact of client movement, the client 12 and the server 40 can (1) 20 use message queues to store communications affected by an interruption for subsequent transmission if communications between the client 12 and the server 40 are resumed; and (2) use bandwidth adaptation techniques to maintain a persistent connection between the client 12 and the server 40 although a route between the client 12 and the server 40 is momentarily lost.

- 68 -

**Message Queues**

The client 12 may register an interest in certain data on server 40. In one embodiment, the data are an object. Objects, as will be appreciated by those skilled in the art, are generally programming units that include data and functionality, and are instances of classes. For  
5 example, the client 12 may be interested in updated information pertaining to particular objects. In one embodiment, server 40 may also include meta-objects, which are objects that have no physical representation, and are classes with methods and attributes that serves as a factory to create new objects. Meta-objects may not be instantiated, (i.e., meta-objects generally do not provide a representation of a physical object). Instead, meta-objects may serve as templates  
10 from which objects that represent physical objects are constructed.

The client 12 maintains local copies of objects on the server 40 and updates these objects, as necessary, when communicating with the server 40. Relevant objects associated with the server 40 may be replicated, (i.e., databases associated with server 40 may be replicated), on the client 12 to provide the client 12 with the local copies of objects. Local copies provide the client  
15 12 with access to relatively up-to-date information should the client 12 move out of the communications range of the subnet 10, interrupting communications with the server 40. While the link 24 is broken, the local copies of the objects, which are active entities, can continue to run on the client 12. Then when the client 12 moves back into the communications range of subnet 10 or the subnet 20 and reestablishes communications with the server 40 over the same or a  
20 different route, the server 40 can provide the client 12 with updated, or current, information. That is, the server 40 may update the local copies of the objects that are present on the client 12 in, for example, a local cache.

In general, because communications between the server 40 and the client 12 are synchronous, the server 40 is aware of all objects that are associated with the client 12. Server  
25 40 may then be able to save state information associated with the client 12. Therefore, server 40

- 69 -

may restore the current state of the client 12 as appropriate (e.g., when lost link 24 is re-established). It should be appreciated, however, that server 40 is generally not aware of any semantics with regards to objects. Rather, the server 40 is only aware that objects have been updated, and, further, that the corresponding updates should be forwarded to the client 12 as  
5 appropriate.

Server 40 includes an object list that is a list of all objects associated with the server 40 and which are to be updated. In other words, the object list is a queue of object updates. The client 12 may communicate substantially with the server 40 after the client 12 is registered with respect to server 40. That is, client 12 may send commands to server 40. In one embodiment,  
10 such commands include lists of topics in which the client 12 is interested. The server 40 may send update messages to the client 12 to indicate that certain objects on the client 12 should be updated such that the states of the objects on the client 12 are consistent with the states of the corresponding objects on the server 40. The client 12 also includes an object list, (i.e., a client object list), that contains substantially all objects that are associated with the client 12. In  
15 general, the new client object list contains all objects, which are associated with the server 40 and which the client is "interested" in.

The client 12 communicates with the server 40 over a route (or path) through the subnet 10 or subnet 20 determined by the routing nodes 14. The client 12 may transmit data to the server 40 directly or through a message queue. The client 12 queues data on the message queue  
20 when, for example, data has been modified and is to be sent to the server 40. Specifically, when the client 12 creates or modifies data, the data is sent to the server 40 through the message queue. The communications between the client 12 and the message queue may, in the described embodiment, be performed using a potentially unreliable communications link (e.g., wireless link), while the communications between the message queue and server 40 are typically more  
25 reliable, (e.g., wired link).



- 70 -

Data is placed on the message queue by the client 12, and is removed from the message queue by the server 40 or, more specifically, communications software associated with server 40. Data is removed from the message queue after the data has been successfully received by the server 40.

- 5           When the client 12 creates data (e.g., objects), the client 12 typically associates that data with a unique identifier that is used by the client 12 and the server 40 to identify that data. One example of a unique identifier is a timestamp. The associated timestamp is updated each time the data are updated or modified by the client 12. A timestamp essentially prevents data conflicts from arising when more than one client attempts to modify that data at a given time.
- 10          Specifically, timestamps are monotonically increasing such that substantially no data conflicts between unique identifiers can arise. Other embodiments use different mechanisms to uniquely identify data, such as an authorization level that is associated with the users of a particular client; a priority level that is associated with the particular type of data, and the order in which the data are received (e.g., LIFO, FIFO).
- 15           Similarly, the server 40 can communicate directly to the client 12 or through a message queue (s), or lists, for storing objects in which the client 12 has indicated an interest. The message queue(s) can be part of or separate from the server 40. Hence, data may be transmitted to the client 12 from the server 40 through such message queues. In other words, the server 40 may use substantially the same heuristics as the client 12 for sending data. Data is placed on the
- 20          message queue by the server 40 and removed from the message queue by the client 12. Again, data is removed from the message queue when the client 12 has successfully received previously removed data.

## **5. Adaptive Use of Network Bandwidth**

- In general, within the subnet 10, which includes wireless links, a variety of different
- 25          failures can interrupt or cause a communications outage. For example, a failure may be due to a



- 71 -

hardware problem at either the client 12 or the server 40. A failure may also be the result of a software problem, (e.g., data may be successfully received but acknowledgement of the receipt may fail). Failures may also occur because of problems with links, as mentioned previously. Such failures may include a failure of any link on a route between the client 12 and the server 40.

5 It should be appreciated that in some cases, more than one failure may occur at any given time.

To adaptively handle interruptions in communications between the client 12 and the server 40, the internetworking system 2 may run diagnostics to determine the cause of the interruption. After the cause is determined, the system 2 makes corrections that restore communications. Depending upon current system parameters, such adaptive corrections include,  
10 but are not limited to attempting (1) to reestablish the same interrupted connection between the client 12 and the server 40, (2) to establish a connection between the client 12 to a redundant server, or (3) to establish a new connection between the client 12 with the server 40. Other techniques used alone or in combination with the aforementioned corrections include varying and/or increasing the waiting period between unsuccessful attempts to establish a connection the  
15 client 12 and the server 40 and adjusting the length of transmitted packets. Such techniques can be used in response to current bandwidth conditions in the subnet 10.

When a "network dropout" occurs in the subnet 10, (e.g., when the client 12 or the server 40 appears to be out of communication with the subnet 10), standard client-server systems, such as those based upon TCP/IP, typically operate under the assumption that the failure is due to  
20 network congestion. As will be understood by those skilled in the art, although a network dropout in a low-bandwidth, wireless subnet may indeed occur as a result of network congestion, the network dropout may also occur for a variety of other reasons including, but not limited to, packet loss due to coverage problems.

Packet loss associated with a network typically involves either the failure of transmission  
25 of a packet of data or the loss of some of the data transmitted in a packet. Although packet

- 72 -

losses can occur for any number of reasons, packet losses often occur when the client 12 is at least temporarily out of range of the subnet 10 or when a communication link in the route between the client 12 and the server 40 is temporarily interrupted.

By counting the number of packets sent and the total number of packets acknowledged, the packet loss in a system may be determined. Measuring packet loss enables the manner in which packets are resent or rebroadcast to be dynamically changed such that the resending of packets is substantially optimized with respect to the network.

The internetworking system 2 can use the measure of packet loss to determine the length of packets that are transmitted between the client and the server 40. For example, if packets with lengths of 1000 bytes experience a 15% packet loss, and packets with lengths of 100 bytes experience a 1% packet loss, then the client 12 and server 40 can tune the length of transmitted packets to minimize the percentage of packets that fail to reach their destination. A factor in determining the packet length is the tradeoff between data throughput and the percentage of packet loss. That is, the smaller the packet length, the greater the percentage of packets that reach their destination, but the lower the percentage of payload (i.e., data) transmitted in each packet because each packet also carries a number of overhead bits.

Also, the client 12 and the server 40 can dynamically adjust the packet length based upon packet loss measurements that are taken periodically. The client 12 and/or the server 40 can make such packet length adjustments. Further, the client 12 can use a packet length that differs from the packet length used by the server 40; packets transmitted from the client 12 to the server 40 may take a different route with different bandwidth capabilities than packets transmitted from the server 40 to the client 12.

When a network dropout occurs due to network congestion, repeated attempts may be made to reconnect a "dropped out" client 12 or server 40 to the subnet 10. If a network dropout occurs due to packet loss and attempts are made to reconnect the client 12 or the server 40 to the

- 73 -

subnet 10, the overall performance of the subnet 10 may degrade to a point where the overall performance of the subnet 10 is unacceptable. That is, attempting to initiate a connection that in fact has not been lost may preclude other connections from being made, thereby preventing the transmission of data which would be made across those connections.

5           Although a variety of different methods may be used to actually determine if a network dropout is the result of network congestion or of packet loss, such a determination may be made using ongoing statistical measurements. Alternatively, the speed at which data is transmitted may be changed. Typically, when a network dropout is due to packet loss, changing the speed of data transmission often solves the network dropout. However, when network dropout is due to  
10 network congestion, changing the speed of data transmission may have no effect and may worsen the throughput.

          In order to enable communications to be optimized to reflect actual network conditions, the client-server system may measure the roundtrip time for packet transmission. That is, the amount of time that elapses while a packet of data is transmitted from the client 12 to the server  
15 40, or vice versa, may be measured. Although the measurements may be used for substantially any purpose, the measurements are often used to characterize the quality of a connection or route between the client 12 and the server 40. By way of example, for certain networks the duration of a roundtrip can indicate whether a connection is good; short roundtrips are associated with good connections, while long roundtrips are associated with poor connections. The measurements of  
20 roundtrip times for a variety of different packets may further be used to statistically determine how long to wait between attempts to resend an unsuccessfully sent packet.

          Fig. 17 shows an embodiment of a process used by the client 12 and the server 40 establish and maintain a persistent connection in a dynamically changing network environment using the above-described bandwidth adaptation techniques. Although the process is described  
25 generally from the perspective of the client 12 sending messages to the server 40, the process

- 74 -

also applies to when the server 40 sends messages to the client 12. The process begins (step 350) by establishing communications between the client 12 and the server 40. Attempts to establish a connection with the server 40 can begin when the client 12 comes within range of the subnet 10. The client 12 sends a packet and awaits a reply from the server 40. The client 12 then waits a  
5 specified period of time. If that period elapses without a receiving a response, the client 12 attempts again to establish a connection with the server 40 by sending another packet. Again, the client 12 waits a specified period of time, but the current waiting period is longer than the previous waiting period. By waiting for a longer period (i.e., "backing off") on the subsequent connection attempt, the client 12 is accommodating the dynamic and intermittent quality of  
10 mobile wireless networks by giving any response from the server 40 additional time to arrive at the client 12. If the new waiting period also times out, the client 12 sends the packet to the server 40 yet again and waits a still longer period for the reply from the server 40 that establishes the connection.

Under some circumstances, numerous clients 12 (e.g., 200), may arrive within range of  
15 the subnet 10 simultaneously, each attempting to establish a connection with the server 40. For example, consider a military "invasion" scenario in which each participant is equipped with a portable computer capable of establishing a wireless connection to the subnet 10 and thus of communicating with the server 40. These computers are used to coordinate the military invasion and to assist in pinpointing the position of each individual during the operation. An onslaught of  
20 connection attempts could overwhelm the subnet 10 and the server 40 with packets such that only a portion of the computers are able to successfully establish a connection with the server 40. If each computer then backed off for approximately the same period of time before attempting again to connect to the server 40, the outcome might be the same; namely, another onslaught of connection packets that impedes some of the computers from establishing a connection. Thus, in  
25 one embodiment, the computers are configured so that the back-off period is not the same for

- 75 -

each of computers, causing the attempts to connect to the server 40 to be staggered. That is, some computers 12 wait for longer periods than other computers before sending another connection packet to the server 40.

After communications are established over a route through the subnet 10 that includes  
5 one or more wireless links, the client 12 identifies (step 354) a packet of data that is to be sent to the server 40 as having been sent. After identifying the packet as having been sent, the client 12 transmits (step 358) the packet through the subnet 10 over a route determined by the routing nodes 14. The packet may be queued in a message queue and sent to the server 40 based on prioritization within the message queue. Similarly, in some embodiments, if a packet is being  
10 sent from the server 40 to the client 12, the packet may also be added to a message queue and sent to the client 12 as determined by priorities assigned within the message queue.

When the packet that is sent includes data that is to be updated, the data may be sent in a variety of different forms. That is, within an object based system, when an object is modified, either the entire object may be sent in a packet, or substantially only the changes to the object  
15 may be sent in a packet. By way of example, when an object has a size that is smaller than a predetermined threshold, the entire object is sent in a packet. Alternatively, when the object is larger than that threshold, the updates or changes to that object alone may be sent in a packet, although the entire object may also be sent.

The client 12 then determines (step 362) whether it has received an acknowledgement  
20 from the server 40 indicating that the server 40 received the packet. The client 12 may make the determination after a predetermined amount of time has elapsed. Receipt of the acknowledgment indicates that the packet has been successfully transmitted and received. Hence, the client 12 identifies (step 366) the packet as being successfully sent and received, and the process of sending data is completed.

- 76 -

If the client 12 instead determines that no acknowledgement of the packet has been received, then this is an indication that there may have been a failure in the network that prevented the server 40 from receiving the packet. Such failures may include, but are not limited to, failures such as a failure of the client 12, of the server 40, and a communication link in a route  
5 between the client 12 and the server 40. The failures may also be due to packet loss, and not to a physical failure of any component of the overall system.

When the packet has not been successfully received, then a determination is made (step 370) as to whether a maximum number of resend tries has been exceeded. The maximum number of attempts to send a packet between the client 12 and the server 40 may generally be  
10 widely varied, and is typically determined using statistical models based upon the measured behavior of the overall system. The maximum number of resend tries may be updated at any suitable time during the operation of the overall system. By way of example, the maximum number of resend tries may be calculated and, if necessary, revised, whenever the accumulation of statistical information reaches a certain level.

15 When the client 12 determines that the maximum number of resend tries has not been exceeded, another attempt is made (step 358) to send the packet. As mentioned above, the amount of time to wait between resend tries may be based upon statistical calculations based upon information that includes the average roundtrip time for a transmitted packet. An attempt to resend a packet can be successful when the initial failure in sending the packet was the result  
20 of packet loss.

On the other hand, if it is determined that the maximum number of resend tries has been exceeded, then attempts to send the packet over the potentially existing link are aborted. When the maximum number of resend tries has been exceeded, and acknowledgement of packet receipt still has not been received, then it is likely that there has been an interruption of the  
25 communications link, and that the unsuccessfully sending of data was likely not due to packet



- 77 -

losses. Accordingly, an attempt is made (step 374) to reestablish communications between the client 12 and the server 40.

A determination is then made (step 378) as to whether communications between the client 12 and the server 40 have been successfully reestablished. If the determination is that communications have been successfully reestablished, the packet is sent (step 358).  
Alternatively, when it is determined that communications between the client 12 and the server 40 have not been reestablished, then another attempt is made (step 374) to reestablish communications. The number of attempts to reestablish communications between the client 12 and the server 40 may be limited in some cases. In one embodiment, attempts to re-establish communications may be aborted after a predetermined number of attempts have been reached. In other embodiments, when the number of attempts to re-establish communications is limited, the number of attempts that are made may be substantially dynamically determined based on statistical information gathered during the course of communications between client 12 and the server 40. In still another embodiment, after the number of attempts to establish a connection with the server 40 is reached, the attempts to establish a connection with the client 12 can continue with a different server upon which the data are replicated.

By limiting the number of times attempts made to send data and, further, by not first attempting to re-establish communications which may not actually have been interrupted, the amount of available communications bandwidth in a system may be substantially optimized. The bandwidth may be allocated to making actual connections which are required, rather than wasting the bandwidth by immediately attempting to re-establish communications when such re-establishment is not necessary.

In one embodiment, when an attempt is made to send data from the client 12 to the server 40, the data is queued on a message queue such that the data is prioritized for transmission to the server. Generally, a single message queue may be shared between multiple servers.



- 78 -

Figs. 18A and 18B are a diagrammatic representation of the updating of an embodiment of a message queue 380 in accordance with an embodiment of the invention. As mentioned above, when data are created or modified, a timestamp accompanying the data is set or modified, respectively. A message queue 380 is shown at time t3 as including objects that were previously  
5 modified and have not yet been accepted by, (i.e., sent to and successfully received by), the server 40. At time t3, at the head 382 of the message queue 380 is an object "obj 1" that was modified at time t1, followed by an object "obj 6" that was modified at time t2, and an object "obj 9" that was modified at time t3. In the described embodiment, the queue 380 is prioritized in a first-in-first-out (FIFO) manner, although priority can instead be based on a variety of other  
10 factors.

At time t5, the queue 380 further includes an object "obj 3" that was modified at time t4. Also at time t5, object "obj 6" is being modified such that its corresponding timestamp is updated accordingly. Further, at time t6, an object "obj 4" is modified. In one embodiment, the object "obj 6" that was modified at time t2 is superceded by a version of object "obj 6" that is  
15 updated at time t5. That is, the object "obj 6" at timestamp t2 has been replaced with the object "obj 6" at timestamp t5. As shown in Fig. 18B, at time t6, the message queue 380 no longer includes the object "obj 6" that was modified at time t2 and, instead, includes object "obj 6" that was modified at time t5. Within the queue 380, object "obj 6" in one embodiment does not take the priority of object "obj6" at time t2, which has been removed. Instead, object "obj 6" takes a  
20 chronological position within the message queue 380, after object "obj 3" and before after the modification at time t5 of object "obj 6". A variety of techniques for prioritizing objects within the message queue 380 is described in co-pending patent application entitled "Method and Apparatus for Updating Information in a Low-Bandwidth Client/Server Object-Oriented System", serial no. 09/518,753, which is incorporated by reference herein in its entirety for all  
25 purposes.

- 79 -

While the invention has been shown and described with reference to specific preferred embodiments, it should be understood by those skilled in the art that various changes in form and detail may be made therein without departing from the spirit and scope of the invention as defined by the following claims. For example, although the described embodiments illustrate the principles of the invention with respect to wireless networks, such also apply to wire-line  
5 networks.

- 80 -

## APPENDIX A

Network-Level Procedures

The notation LSU(update\_list) represents a link-state-update message that includes the updates (u, v, c, sn) in the update\_list.

```

5      Process_Update(i, nbr, in_message){
        // Called when an update message in_message is received from nbr.
        Update_Topology_Table(i, nbr, in_message, update_list).
        Update_Parents(i).
        For each node src in TT_i {
10          Let update_list(src) consist of all tuples (k, l, c, sn) in update_list such that
            k = src.
            If update_list(src) is nonempty
              Send message LSU(update_list(src)) to children_i(src).} }

        Update_Topology_Table(i, nbr, in_message, update_list){
15          Set update_list to empty list.
          For each ((u,v,c,sn) in in_message) {
            If (p_i(u) == nbr) {
              If ((u,v) is in TT_i and sn > TT_i(u,v).sn) {
                Add (u,v,c,sn) to update_list.
20                Set TT_i(u,v).sn = sn.
                Set TT_i(u,v).c = c.
                If (sn > sn_i(u)) Set sn_i(u) = sn.}
              If ((u,v) is not in TT_i) {
                Add (u,v,c,sn) to TT_i.
25                Add (u,v,c,sn) to update_list.
                If (sn > sn_i(u)) Set sn_i(u) = sn.} } } }

        Link_Change(i,j){
          // Called when the cost of link (i,j) changes.
          If (|TT_i(i,j).c - cost(i,j)|/TT_i(i,j).c > epsilon) {
30            Set TT_i(i,j).c = cost(i,j).
            Set TT_i(i,j).sn = current time stamp SN_i.
            Set update_list = {(i, j, TT_i(i, j).c, TT_i(i, j).sn)
            Send message LSU(update_list) to children_i(i).} }

        Link_Down(i,j){
35          // Called when link (i,j) goes down.
          Remove j from N_i.
          Set TT_i(i,j).c = infinity.
          Set TT_i(i,j).sn = current time stamp SN_i.
          Update_Parents(i).
40          For each (node src in TT_i) remove j from children_i(src).
          Set update_list = {(i,j, infinity, TT_i(i,j).sn)}.
          Send message LSU(update_list) to children_i(i).}

```

- 81 -

```

Link_Up(i,j){
    // Called when link (i,j) comes up.
    Add j to N_i.
    Set TT_i(i,j).c = cost(i,j).
    5   Set TT_i(i,j).sn = current time stamp SN_i.
        Update_Parents(i).
        Set update_list = {(i, j, TT_i(i,j).c, TT_i(i,j).sn)}.
        Send message LSU(update_list) to children_i(i).}

Update_Parents(i){
    10   Compute_New_Parents(i)
        For each (node k in N_i){
            Set cancel_src_list(k), src_list(k), and sn_list(k) to empty.}
        For each (node src in TT_i such that src != i){
            If (new_p_i(src) != p_i(src)){
                15   If (p_i(src) != NULL){
                    Set k = p_i(src).
                    Add src to cancel_src_list(k).}
                Set p_i(src) = new_p_i(src).
                If (new_p_i(src) != NULL){
                    20   Set k = new_p_i(src).
                        Add src to src_list(k).
                        Add sn_i(src) to sn_list(k).}}}
        For each (node k in N_i){
            If (src_list(k) is nonempty){
                25   Send message NEW PARENT(src_list(k), sn_list(k)) to k.}
            If (cancel_src_list(k) is nonempty){
                Send message CANCEL PARENT(cancel_src_list(k)) to k.}}}

Compute_New_Parents(i){
    30   For each (node src in TT_i such that src != i){
        Set new_p_i(src) = NULL.}
        Compute min-hop paths using Dijkstra.
        For each (node src in TT_i such that src != i){
            Set new_p_i(src) equal to the neighbor of node i along the minimum-hop
            path from i to src.}}}

    35   Process_New_Parent(i, nbr, src_list, sn_list){
        // Called when node i receives a NEW PARENT(src_list, sn_list) message from
        nbr.
        Set update_list to empty list.
        For each (node src in src_list) {
            40   Let sn_list.src denote the sequence number corresponding to src in sn_list.
                Add nbr to children_i(src).
                Set new_updates = {(k, l, c, sn) in TT_i such that k = src and sn >
                sn_list.src}.
                Add new_updates to update_list.}
            45   Send message LSU(update_list) to nbr.}

```

- 82 -

```

Process_Cancel_Parent(i,nbr,src_list){
    // Called when node i receives a CANCEL PARENT(src_list) message from nbr.
    For each (node src in src_list) remove nbr from children_i(src).}

Send_Periodic_Updates(i){
5     Set update_list to empty.
    For each (j in N_i such that TT_i(i,j).c != infinity){
        Set TT_i(i,j).sn = current time stamp SN_i.
        Add (i, j, TT_i(i,j).c, TT_i(i,j).sn) to update_list. }
    Send message LSU(update_list) to children_i(i).}

10 Compute_New_Parents2(i){
    S ← ∅;
    For each (v ∈ TT_i) {
        Set d(v) = infinity;
        Set pred(v) = NULL;
15     Set new_p_i(v) = NULL; }
    d(i) ← 0;
    While (there exists w ∈ TT_i - S such that d(w) < infinity){
        Set u = node w ∈ TT_i - S that minimizes d(w);
        Set S = S ∪ {u};
20     For each (v such that (u, v) ∈ TT_i) {
        If (d(u) + 1 < d(v) or [d(u) + 1 = d(v) and new_p_i(u) = p_i(v)]) {
            Set d(v) = d(u) + 1;
            Set pred(v) = u;
            If (u = i) Set new_p_i(v) = v;
25     Else Set new_p_i(v) = new_p_i(u); } } } }

```

### Partial-Topology 1

The function Mark\_Special\_Links() is called whenever the parent  $p_i(src)$  or the set of children  $children_i(src)$  for any source  $src$  changes. The notation  $LSU(update\_list)$  represents a link-state-update message that includes the updates  $(u, v, c, sn, sp)$  in the  $update\_list$ , where  $sp$  is

30 a single bit that indicates whether the link is “special”, i.e., whether it should be broadcast to all nodes.

```

Mark_Special_Links(i){
    For all (outgoing links (i,j)) {Set TT_i(i,j).sp = 0;}
    For all (nodes src != i){
35     if (p_i(src) != NULL and p_i(src) != src){
        Set TT_i(i, p_i(src)).sp = 1;} //Link is special.
        For all (nodes j in children_i(src)){
            Set TT_i(i,j).sp = 1;} //Link is special.
    }
}

```

- 83 -

```

    }

Update_Topology_Table(i, nbr, in_message, update_list){
    Set update_list to empty list.
    For each ((u,v,c,sn,sp) in in_message) {
5        If (p_i(u) = nbr) {
            If ((u,v) is in TT_i and sn > TT_i(u,v).sn) {
                Set TT_i(u,v).sn = sn.
                Set TT_i(u,v).c = c.
                Set TT_i(u,v).sp = sp.
10                (Only links marked as special are forwarded.)
                If (sp = 1) Add (u,v,c,sn,sp) to update_list.
                If (sn > sn_i(u)) Set sn_i(u) = sn.}
            If ((u,v) is not in TT_i) {
                Add (u,v,c,sn,sp) to TT_i.
15                If (sp = 1) Add (u,v,c,sn,sp) to update_list.
                If (sn > sn_i(u)) Set sn_i(u) = sn.}}}}

Process_Update(i, nbr, in_message){
    // Called when an update message in_message is received from nbr.
20    Update_Topology_Table(i, nbr, in_message, update_list).
    Update_Parents(i).
    Mark_Special_Links(i).
    For each node src in TT_i {
        Let update_list(src) consist of all tuples (k, l, c, sn, sp) in update_list such
25        that k = src.
        If update_list(src) is nonempty
            Send message LSU(update_list(src)) to children_i(src).}}

Link_Change(i,j){
    // Called when the cost of link (i,j) changes.
30    If (|TT_i(i,j).c - cost(i,j)|/TT_i(i,j).c > epsilon) {
        Set TT_i(i,j).c = cost(i,j).
        Set TT_i(i,j).sn = current time stamp SN_i.
        Set update_list = {(i, j, TT_i(i, j).c, TT_i(i, j).sn, TT_i(i,j).sp)}.
        Send message LSU(update_list) to children_i(i).}}

35    Link_Down(i,j){
        // Called when link (i,j) goes down.
        Remove j from N_i.
        Set TT_i(i,j).c = infinity.
        Set TT_i(i,j).sn = current time stamp SN_i.
40        Update_Parents(i).
        For each (node src in TT_i) remove j from children_i(src).
        Mark_Special_Links(i).
        Set update_list = {(i,j, infinity, TT_i(i,j).sn, TT_i(i,j).sp)}.
        Send message LSU(update_list) to children_i(i).}

45    Link_Up(i,j){
        // Called when link (i,j) comes up.

```

- 84 -

```

Add j to N_i.
Set TT_i(i,j).c = cost(i,j).
Set TT_i(i,j).sn = current time stamp SN_i.
Update_Parents(i).
5 Mark_Special_Links(i).
Set update_list = {(i, j, TT_i(i,j).c, TT_i(i,j).sn, TT_i(i,j).sp)}.
Send message LSU(update_list) to children_i(i).}

Update_Parents(i){
  Compute_New_Parents(i).
10 For each (node k in N_i)
    Set cancel_src_list(k), src_list(k), and sn_list(k) to empty.
  For each (node src in TT_i such that src != i){
    If (new_p_i(src) != p_i(src)){
      If (p_i(src) != NULL){
15 Set k = p_i(src).
        Add src to cancel_src_list(k).}
      Set p_i(src) = new_p_i(src).
      If (new_p_i(src) != NULL){
        Set k = new_p_i(src).
20 Add src to src_list(k).
        Add sn_i(src) to sn_list(k).}}}
  For each (node k in N_i){
    If (src_list(k) is nonempty){
      Send message NEW PARENT(src_list(k), sn_list(k)) to k.}
25 If (cancel_src_list(k) is nonempty{
      Send message CANCEL PARENT(cancel_src_list(k)) to k.}}}

Compute_New_Parents(i){
  For each (node src in TT_i such that src != i){
    Set new_p_i(src) = NULL.}
30 Compute min-hop paths using Dijkstra.
  For each (node src in TT_i such that src != i){
    Set new_p_i(src) equal to the neighbor of node i along the minimum-hop
    path from i to src.}}

Process_New_Parent(i; nbr, src_list, sn_list){
35 //Called when node i receives a NEW PARENT(src_list, sn_list) message from
    nbr.
  Set update_list to empty list.
  For each (node src in src_list) {
    Let sn_list.src denote the sequence number corresponding to src in sn_list.
40 Add nbr to children_i(src).
    If (src != i) Set TT_i(i, nbr).sp = 1. //Link to nbr is special.
    If (src = i) Set new_updates = {(src, v, c, sn, sp) in TT_i such that
      sn > sn_list.src}.
    If (src != i) Set new_updates = {(src, v, c, sn, sp) in TT_i such that
45 sn > sn_list.src and sp = 1}. //Only special links are sent.
    Add new_updates to update_list.}

```



- 85 -

Send message LSU(update\_list) to nbr.}

Process\_Cancel\_Parent(i,nbr,src\_list){

// Called when node i receives a CANCEL PARENT(src\_list) message from nbr.

For each (node src in src\_list) remove nbr from children\_i(src).

5 Mark\_Special\_Links(i). }

Send\_Periodic\_Updates(i){

Set update\_list to empty.

For each (j in N\_i such that TT\_i(i,j).c != infinity){

Set TT\_i(i,j).sn = current time stamp SN\_i.

10 Add (i, j, TT\_i(i,j).c, TT\_i(i,j).sn, TT\_i(i,j).sp) to update\_list. }

Send message LSU(update\_list) to children\_i(i).}

### Partial-Topology 2

Update(i, k, in\_message){

Update\_Topology\_Table(i, k, in\_message);

15 Lex\_Dijkstra; // Uses lexicographic Dijkstra to compute T<sub>i</sub>

Generate\_Updates(i, update\_list);

if (k does not equal i and update\_list is non-empty){

Send\_Updates\_Children(i, update\_list);

Update\_Parents(i);

20 }

Send\_Updates\_Children(i, update\_list){

For each (node k ∈ N<sub>i</sub>) {out\_message(k) ← 0;}

For each (node src ∈ TT\_i s.t. src does not equal i){

update\_list(src) ← {(k, l, c) ∈ update\_list s.t. k = src};

25 for each (node k ∈ children\_i(src)){

Add update\_list(src) to out\_message(k);}

}

For each (node k ∈ N<sub>i</sub> s.t. out\_message(k) is non-empty){

Send the message out\_message(k) to node k;}

30 }

Update\_Topology\_Table(i, k, in\_message){

For each ((u, v, c) ∈ in\_message{

// Process only updates received from the parent p<sub>i</sub>(u)

if (p<sub>i</sub>(u) = k or k = i){

35 if ((u, v) ∉ TT\_i or c! = TT\_i(u, v).c{

TT\_i(u, v) ← (u, v, c);

Mark (u, v) as changed in TT\_i;}

}

}

40 if (in\_message is a PARENT\_RESPONSE){

For each (u such that in\_message includes source u){

if (p<sub>i</sub>(u) = k and pending\_i(u) = 1){

pending\_i(u) = 0;

- 86 -

```

For each ( $v$  such that  $TT_i$  contains an entry for  $(u, v)$ ) {
    if ( $in\_message$  does not contain update for link  $(u, v)$ ) {
         $TT_i(u, v).c \leftarrow \infty;$ 
// indicates link should be deleted
Mark  $(u, v)$  as changed in  $TT_i;$ 
    }
}

Process_Cancel_Parent( $i, nbr, src\_list$ ) {
    For each ( $src \in src\_list$ )
        children_ $i$ ( $src$ )  $\leftarrow$  children_ $i$ ( $src$ ) - { $nbr$ };
}

Generate_Updates( $i, update\_list$ ) {
    update_list  $\leftarrow$  0;
    for each (entry  $(u, v, c, c') \in TT_i$ ) {
        if (( $u, v$ ) is in new  $T_i$  and (( $u, v$ ) is marked as changed or is not in old  $T_i$ )) {
            Add  $(u, v, c)$  to update_list;
             $T_i(u, v).c' \leftarrow T_i(u, v).c;$ 
             $R_i \leftarrow R_i \cup \{(u, v)\};$ 
        }
        else if (( $u, v$ ) is in  $R_i$  but not in new  $T_i$  and  $c > c'$ ) {
            Add  $(u, v, \infty)$  to update_list; // delete update
             $T_i(u, v).c' \leftarrow \infty;$ 
            Remove  $(u, v)$  from  $R_i;$ 
        }
        if ( $TT_i(u, v).c = \infty$ )
            Remove  $(u, v)$  from  $TT_i;$ 
    }
}

Update_Parents( $i$ ) {
    For each (node  $k \in N_i$ ) {
        cancel_src_list( $k$ )  $\leftarrow$  0;
        src_list( $k$ )  $\leftarrow$  0;}
    For each (node  $src \in TT_i$  such that  $src \neq i$ ) {
        new_p_ $i$ ( $src$ )  $\leftarrow$  next node on shortest path to  $src;$ 
        if (new_p_ $i$ ( $src$ )  $\neq p_i(src)$ ) {
            if (new_p_ $i$ ( $src$ )  $\neq NULL$ ) {
                 $k \leftarrow p_i(src);$ 
                cancel_src_list( $k$ )  $\leftarrow$  cancel_src_list( $k$ )  $\cup$  { $src$ };
            }
        }
    }
}

```

- 87 -

```

        if (new_p_i(src) ≠ NULL){
            k ← new_p_i(src);
            src_list(k) ← src_list(k) ∪ {src};
        }
5      p_i(src) ← new_p_i(src);
    }
  }
  For each (node k ∈ Ni){
    if (src_list(k) ≠ 0)
10      Send NEW_PARENT(src_list(k)) to node k;
    if (cancel_src_list(k) ≠ 0)
      Send CANCEL_PARENT(cancel_src_list(k)) to node k;
  }
}

15 Process_New_Parent(i, nbr, src_list){
  update_list ← 0;
  for each (node u ∈ u_list) {
    children_i(u) ← children_i(u) ∪ {nbr};
    updates(u) ← {(u, v, c) ∈ TT_i such that (u, v) ∈ Ti};
20    update_list ← update_list ∪ updates (u);
  }
  Send PARENT_RESPONSE(src_list, update_list) to nbr;}

```

- 88 -

## Claims

What is claimed is:

- 1 1. An address format for achieving address compatibility between a previous version and a  
2 new version of Internet Protocol (IP), comprising:  
3 a prefix address portion containing a globally aggregatable IP address prefix; and  
4 an interface identifier portion concatenated to the prefix portion and containing an  
5 embedded IP address assigned according to the previous IP version, the concatenation of  
6 the prefix address portion and the interface identifier portion forming an address that is  
7 compatible with the new IP version and enables packets to be routed to addresses  
8 assigned according to the previous IP version.
- 1 2. The address format of claim 1 wherein the interface identifier portion includes a type  
2 field that indicates whether the interface identifier portion contains an embedded IP  
3 address assigned according to the previous IP version.
- 1 3. The address format of claim 1 wherein the address embedded in the interface identifier  
2 portion can be globally non-unique.
- 1 4. The address format of claim 1 wherein the prefix address portion enables routing nodes  
2 to route packets to a particular node having an IP address that is the same as the  
3 embedded IP address through nodes communicating according to the new version of IP to  
4 that particular node.
- 1 5. The address format of claim 1 wherein the address formed by concatenating the prefix  
2 address portion and the interface identifier portion enables routing nodes to tunnel  
3 packets sent to a particular node assigned that address through one or more nodes  
4 operating according to the previous version of Internet Protocol.

- 89 -

- 1    6.    A method for enabling incremental deployment of Internet Protocol version 6 (IPv6) in a  
2        network having Internet Protocol version 4 (IPv4) nodes, comprising:  
3                associating a globally aggregatable IPv6 address prefix with an IPv4 node having  
4        an IPv4 address and being deployed in the network;  
5                configuring the IPv4 node with an IPv6-IPv4 compatibility address that includes a  
6        prefix portion containing the IPv6 address prefix and a interface identifier portion  
7        containing the IPv4 address of the IPv4 node.
- 1    7.    The method of claim 6 further comprising routing a packet addressed to the IPv4 node  
2        configured with the IPv6-IPv4 compatibility address across IPv6 infrastructure using the  
3        IPv6 prefix portion.
- 4    8.    The method of claim 6 further comprising tunneling a packet addressed to the IPv4 node  
5        configured with the IPv6-IPv4 compatibility address across IPv4 infrastructure using the  
6        IPv4 address embedded in the interface identifier portion.

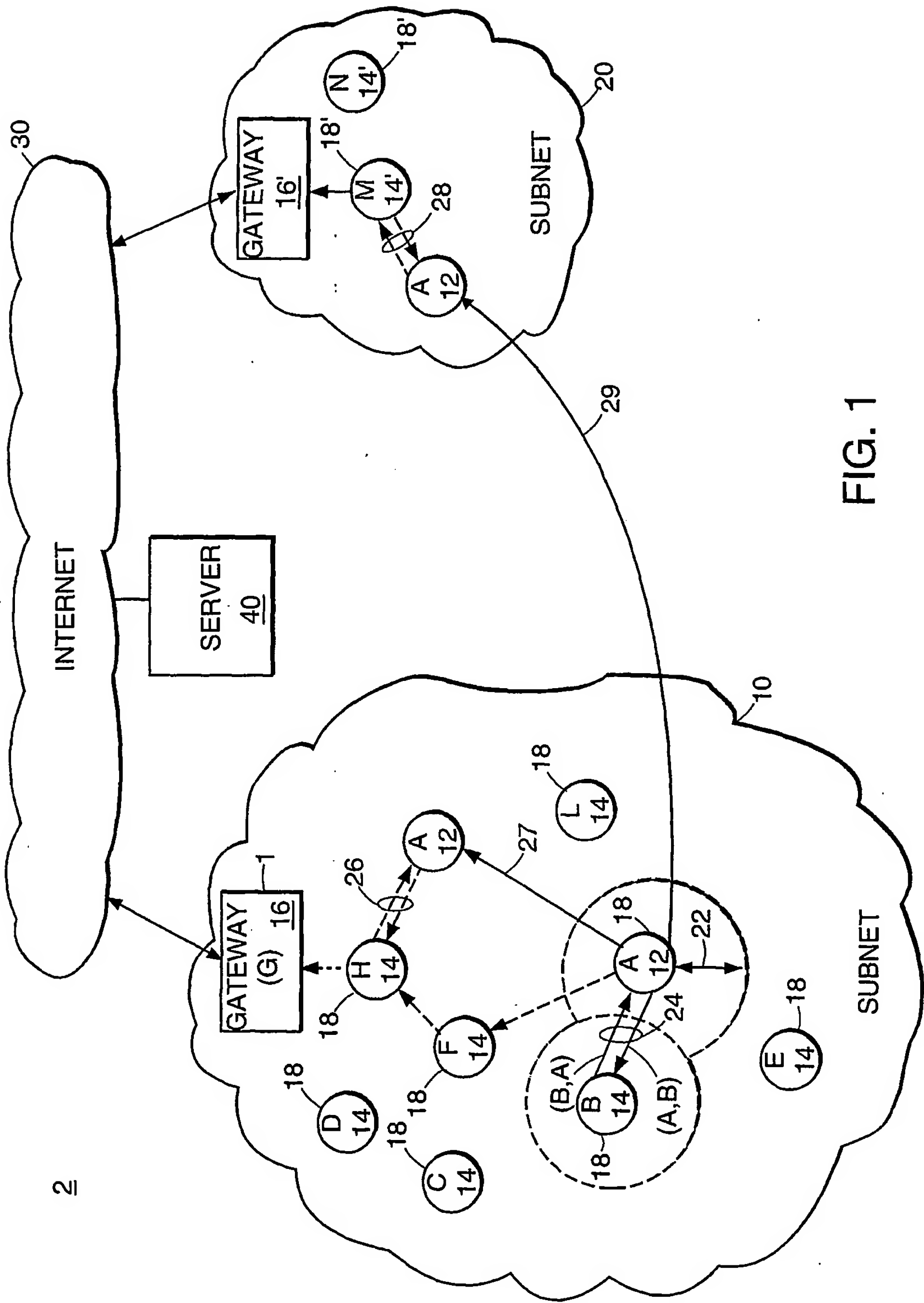


FIG. 1

2/20

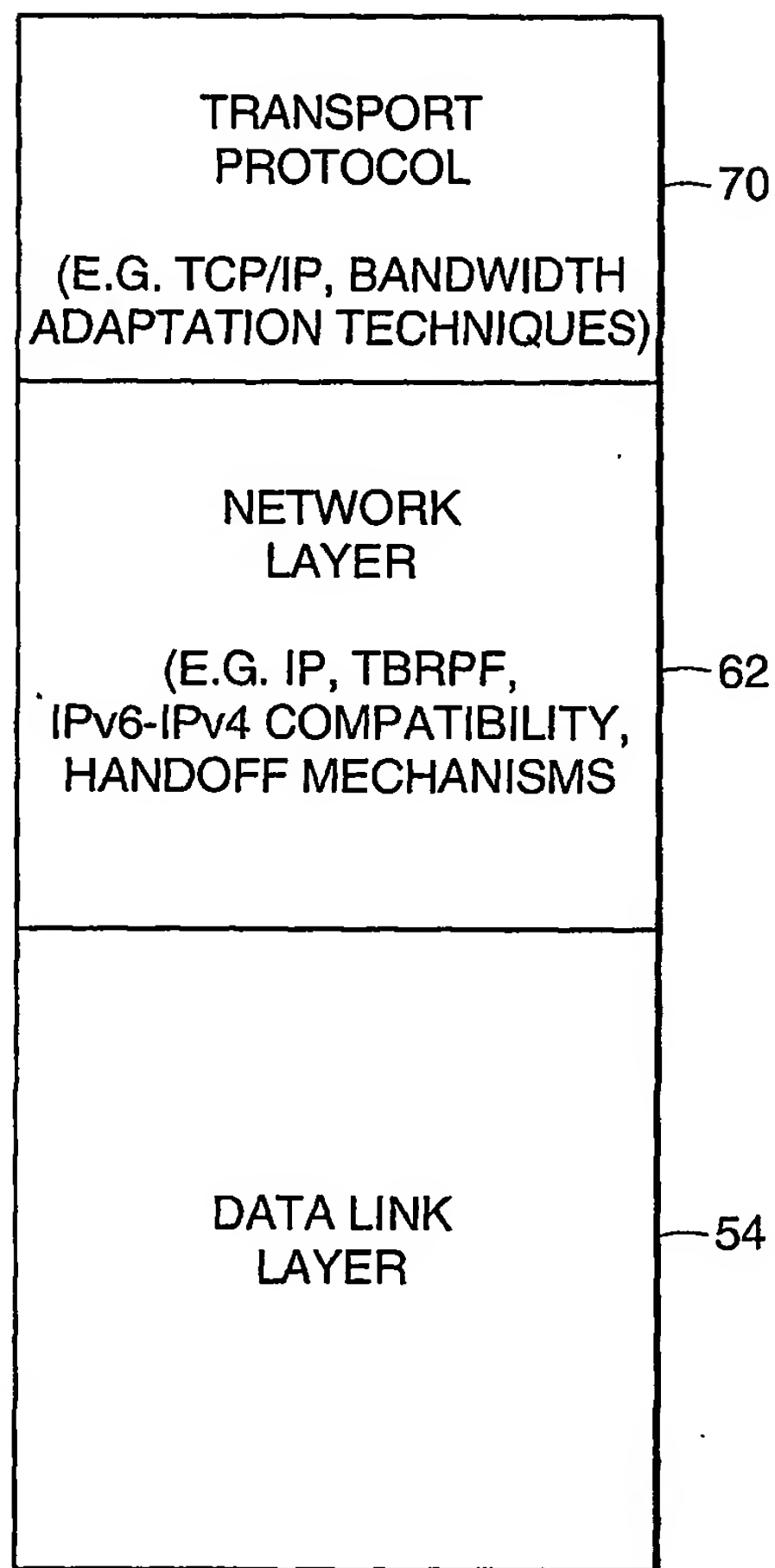
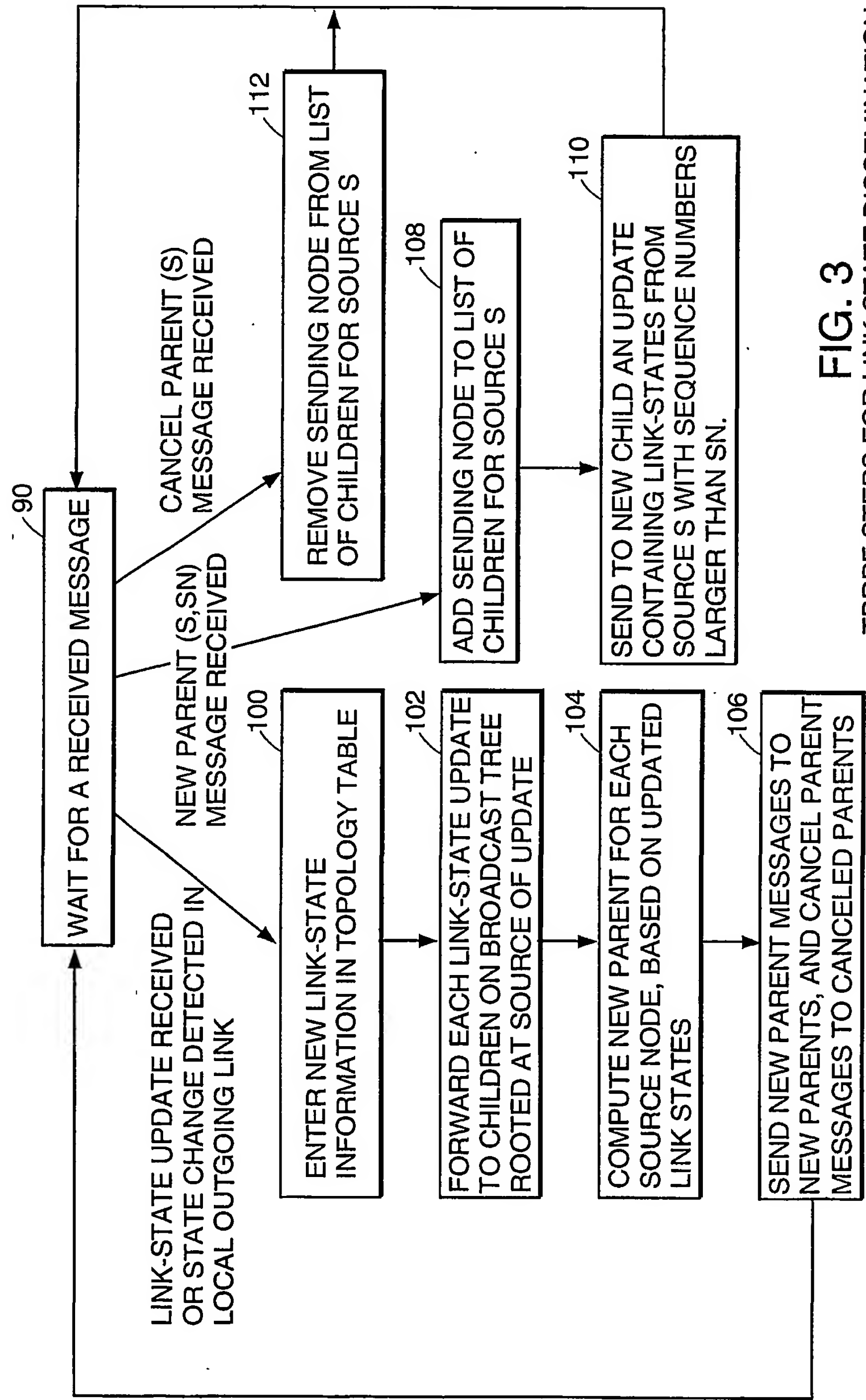


FIG. 2





**FIG. 3**  
TBRPF STEPS FOR LINK-STATE DISSEMINATION  
(NOT INCLUDING NEIGHBOR DISCOVERY)

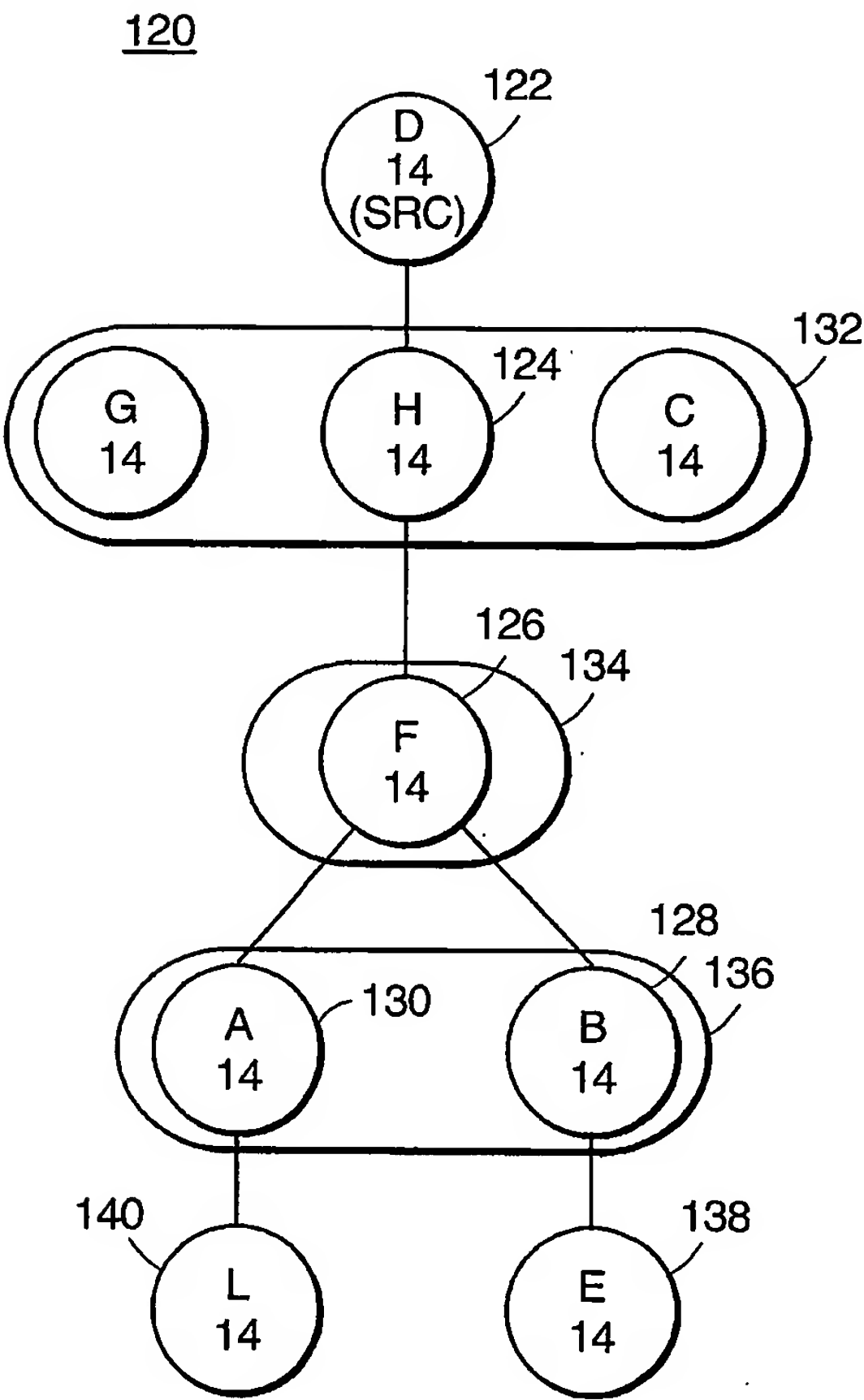


FIG. 4

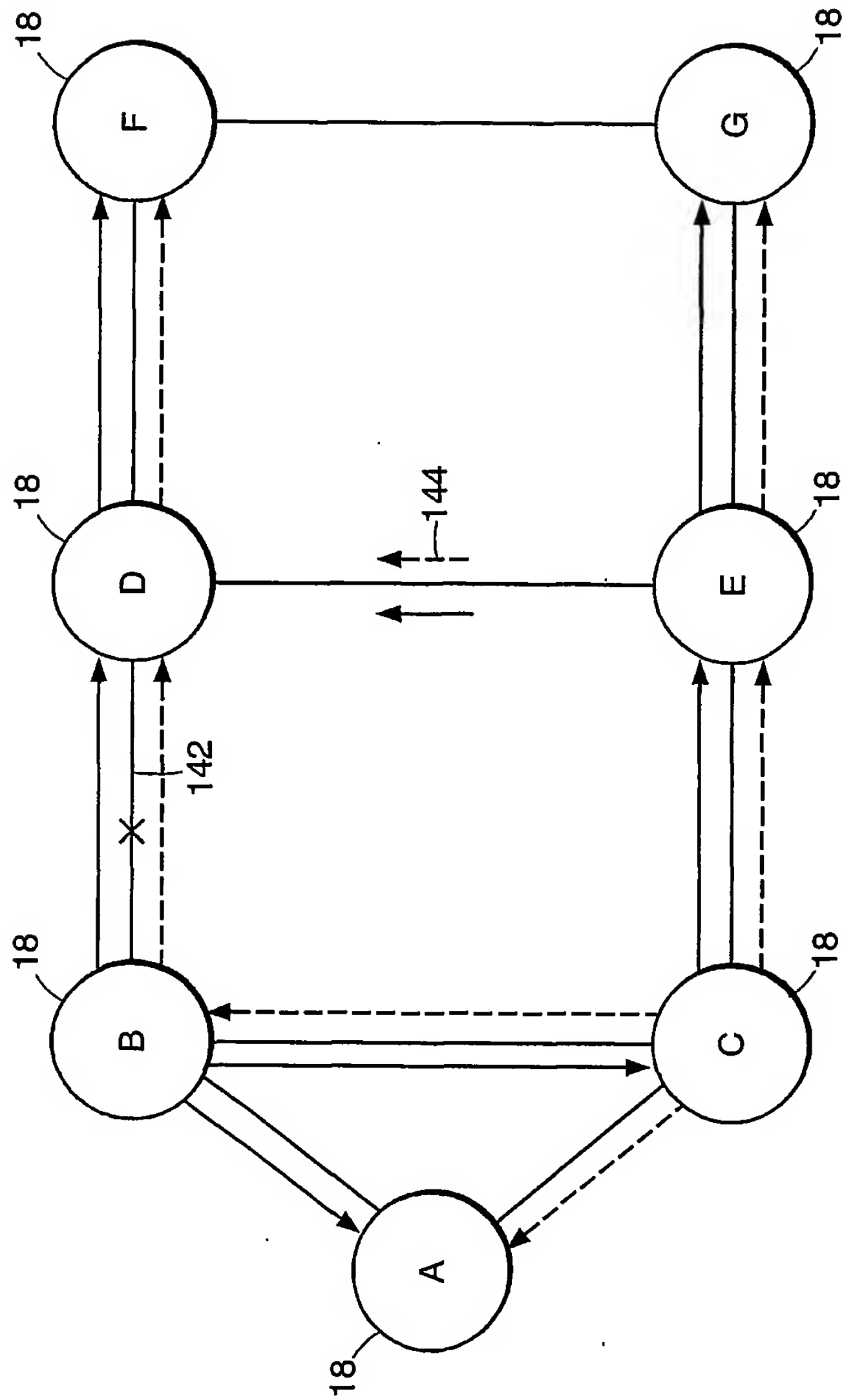
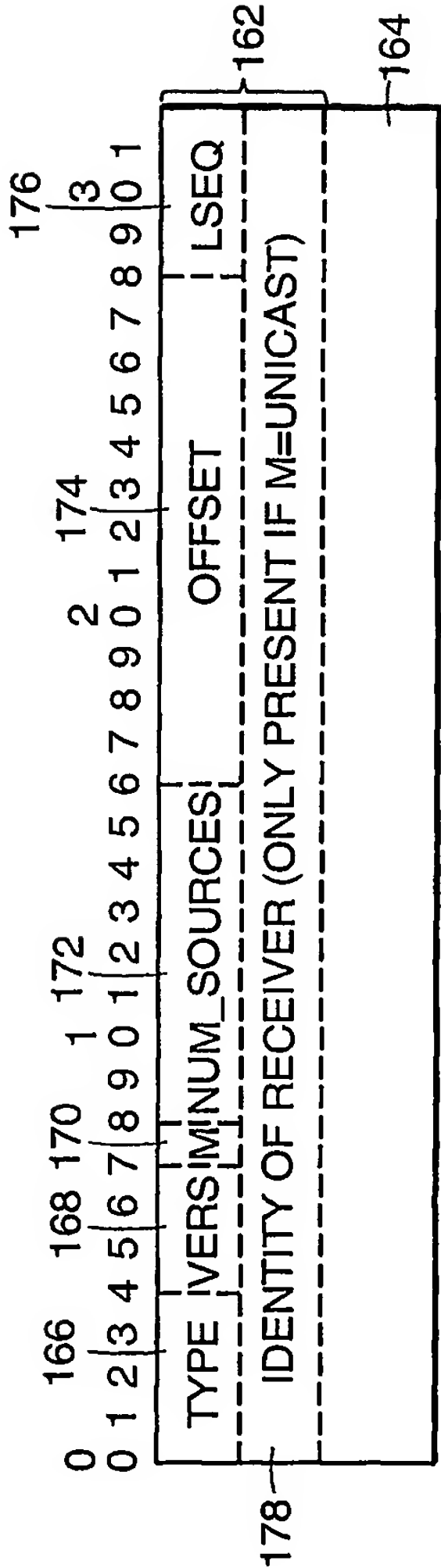


FIG. 5

160



TBRPF MESSAGE HEADER

FIG. 6

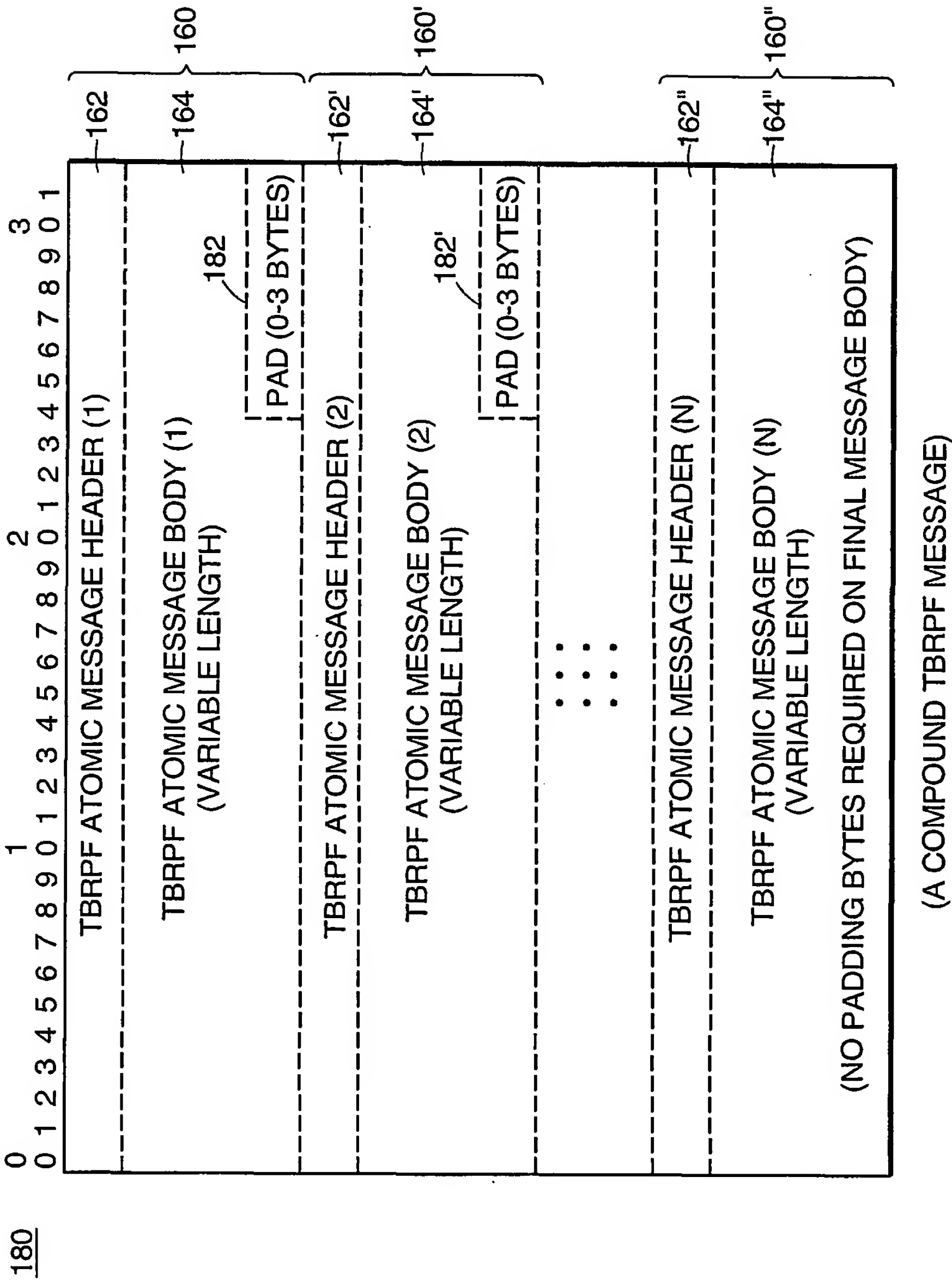
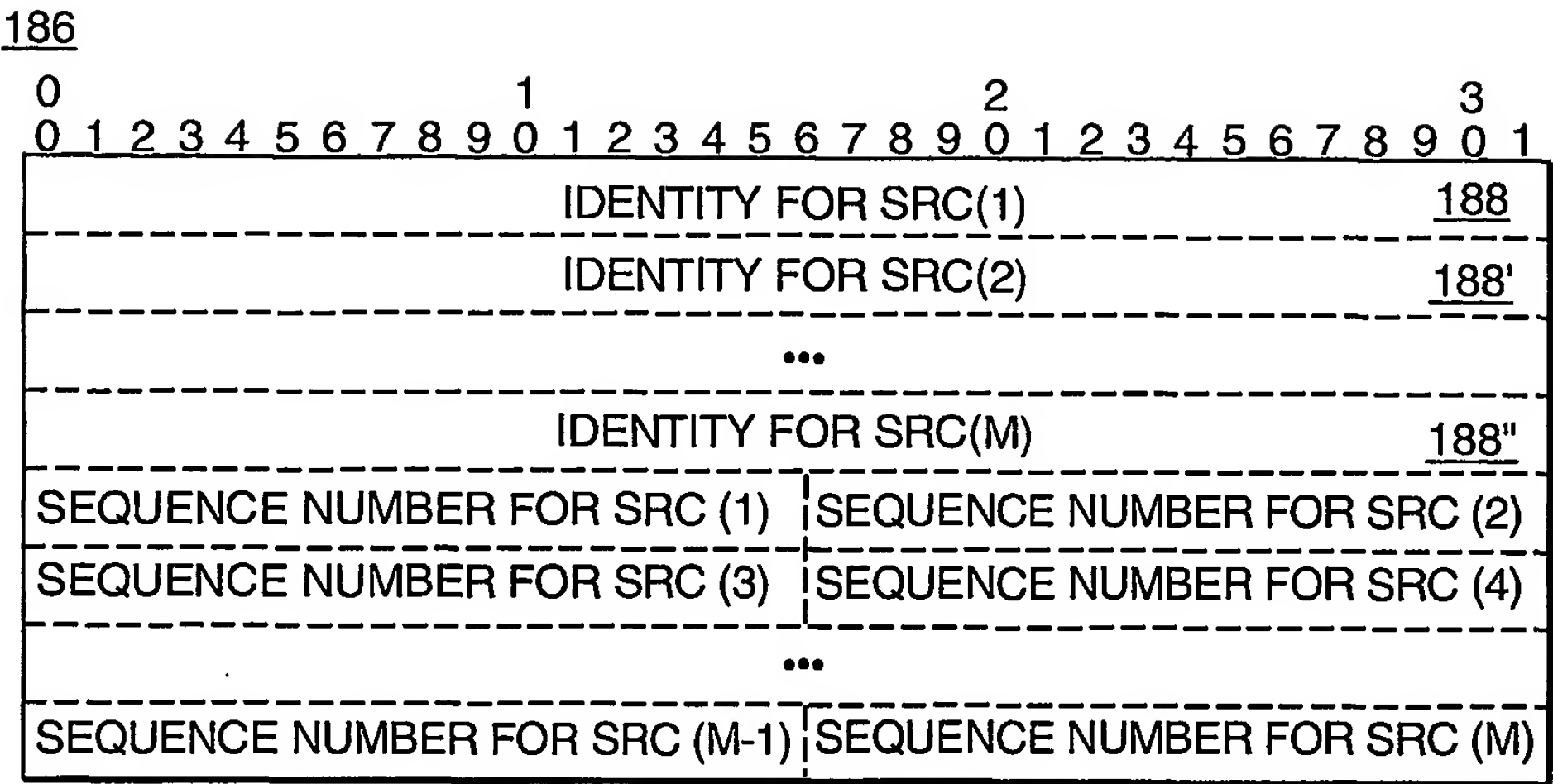


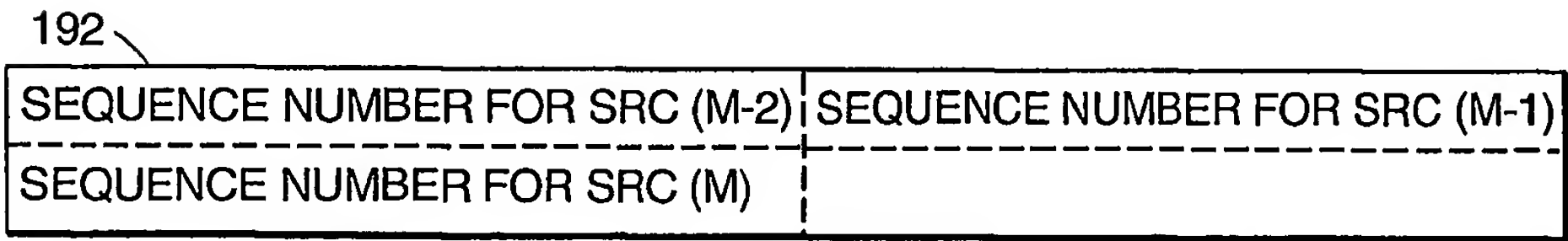
FIG. 7

8/20



NEW PARENT MESSAGE

FIG. 8A

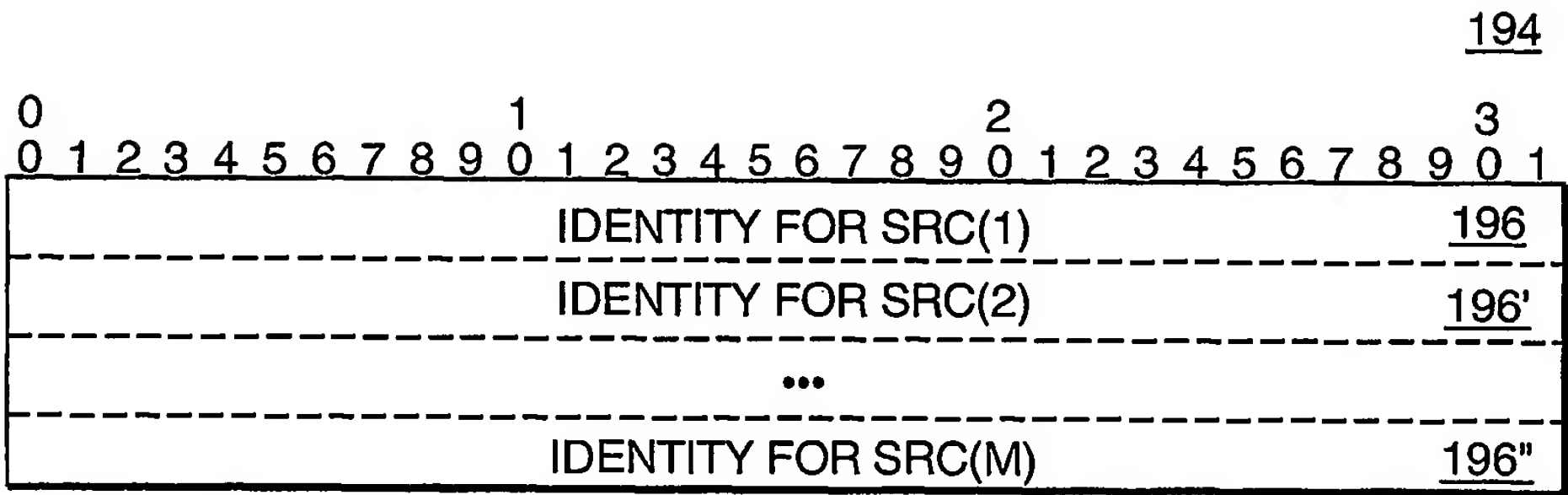


SHOWS AN ALTERNATIVE ENDING THE NEW PARENT MESSAGE

FIG. 8B

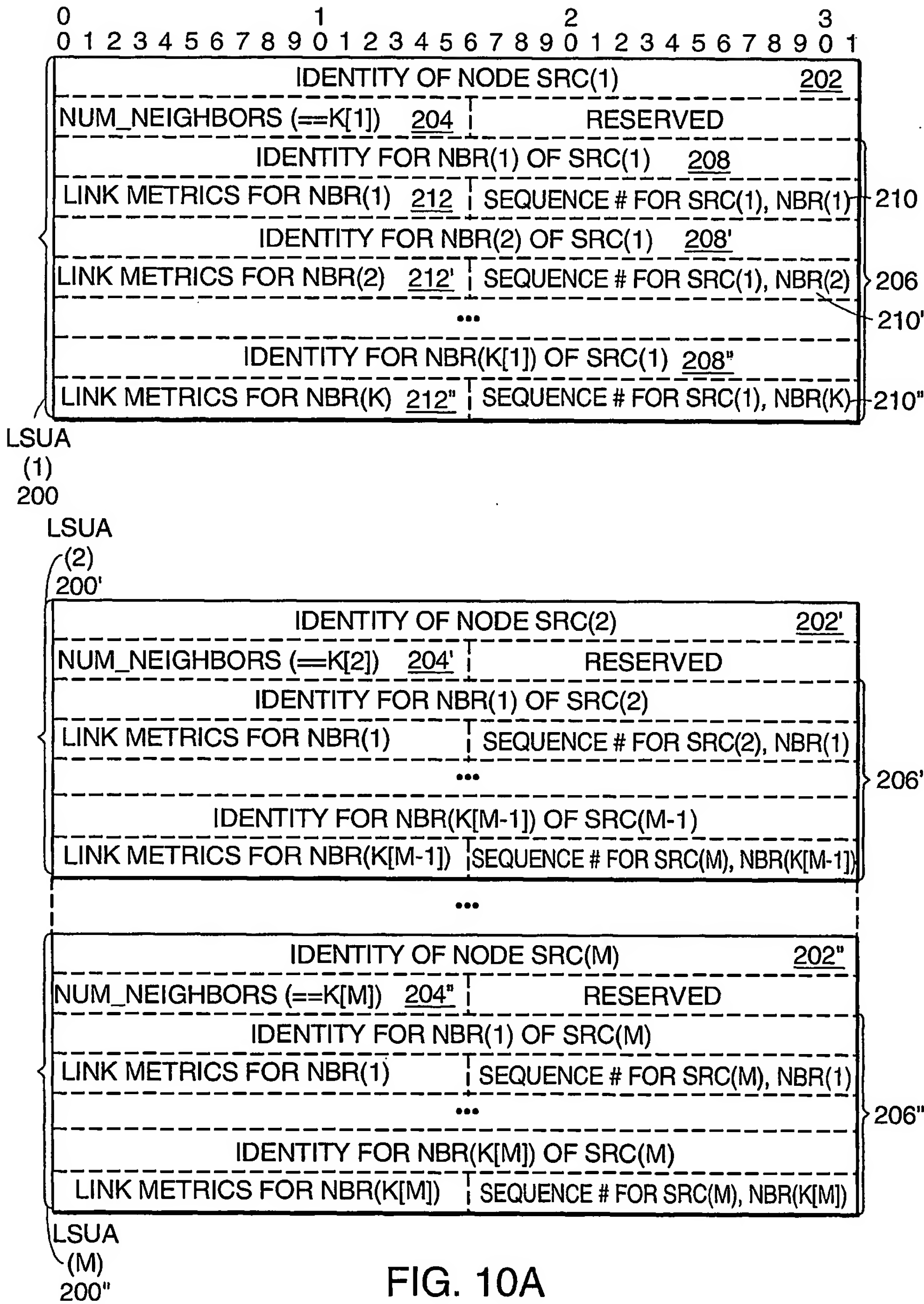


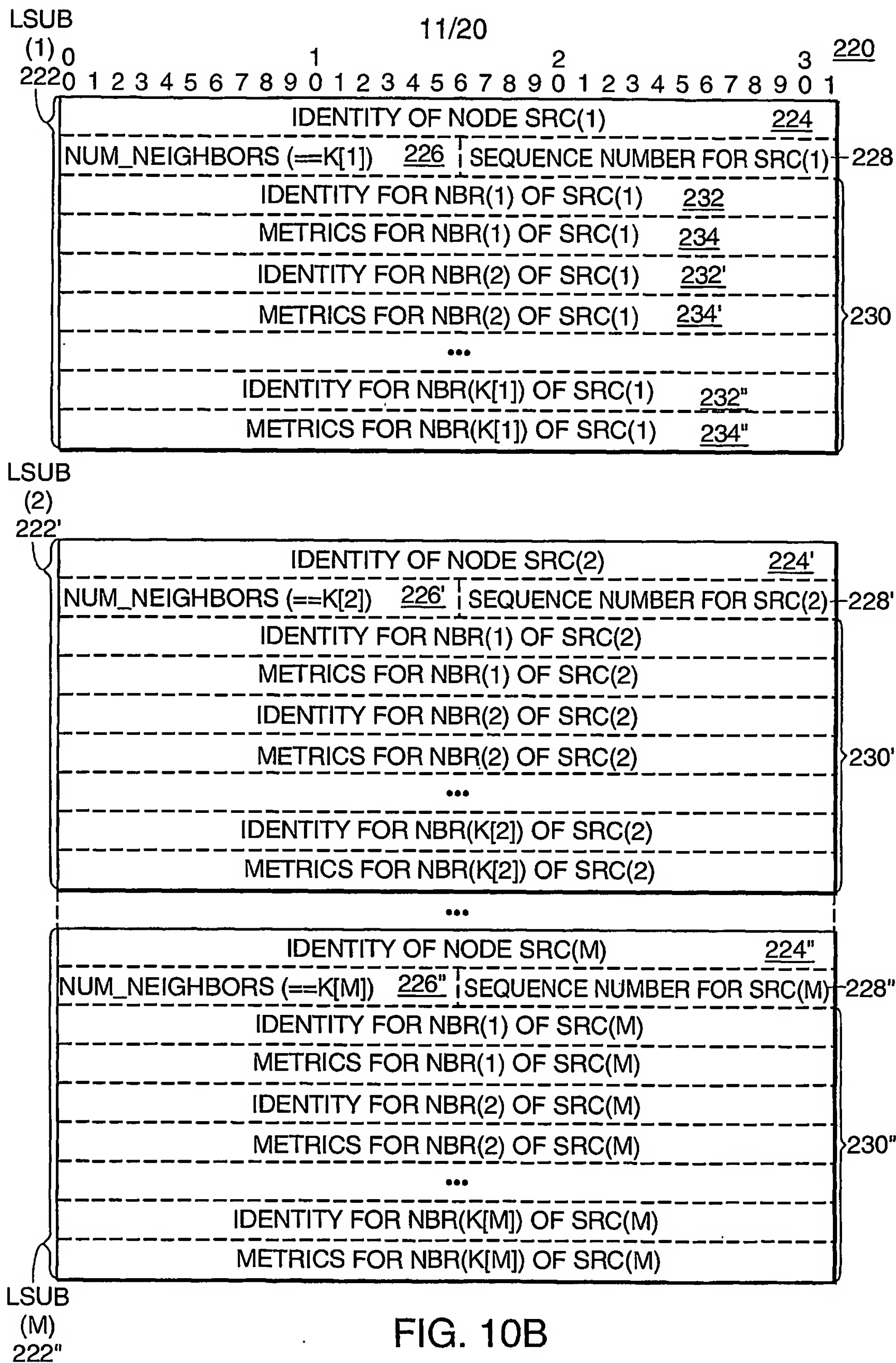
9/20



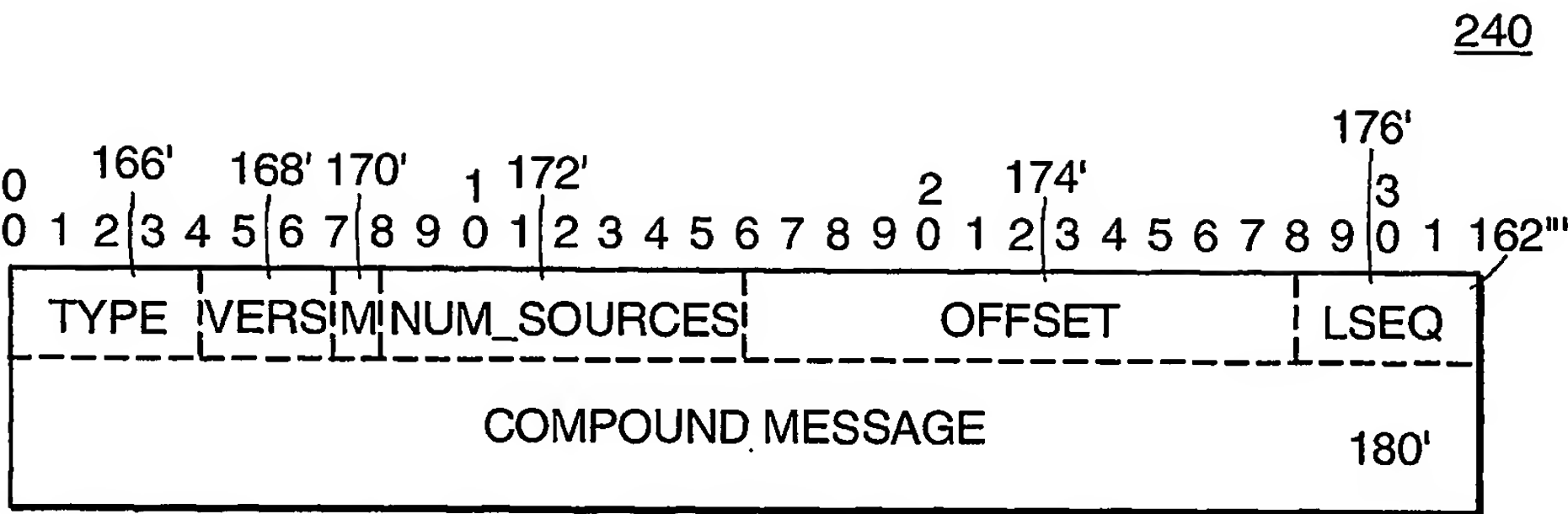
CANCEL\_PARENT MESSAGE  
FIG. 9

10/20





12/20



RETRANSMISSION\_OF\_BROADCAST MESSAGE

FIG. 11

13/20

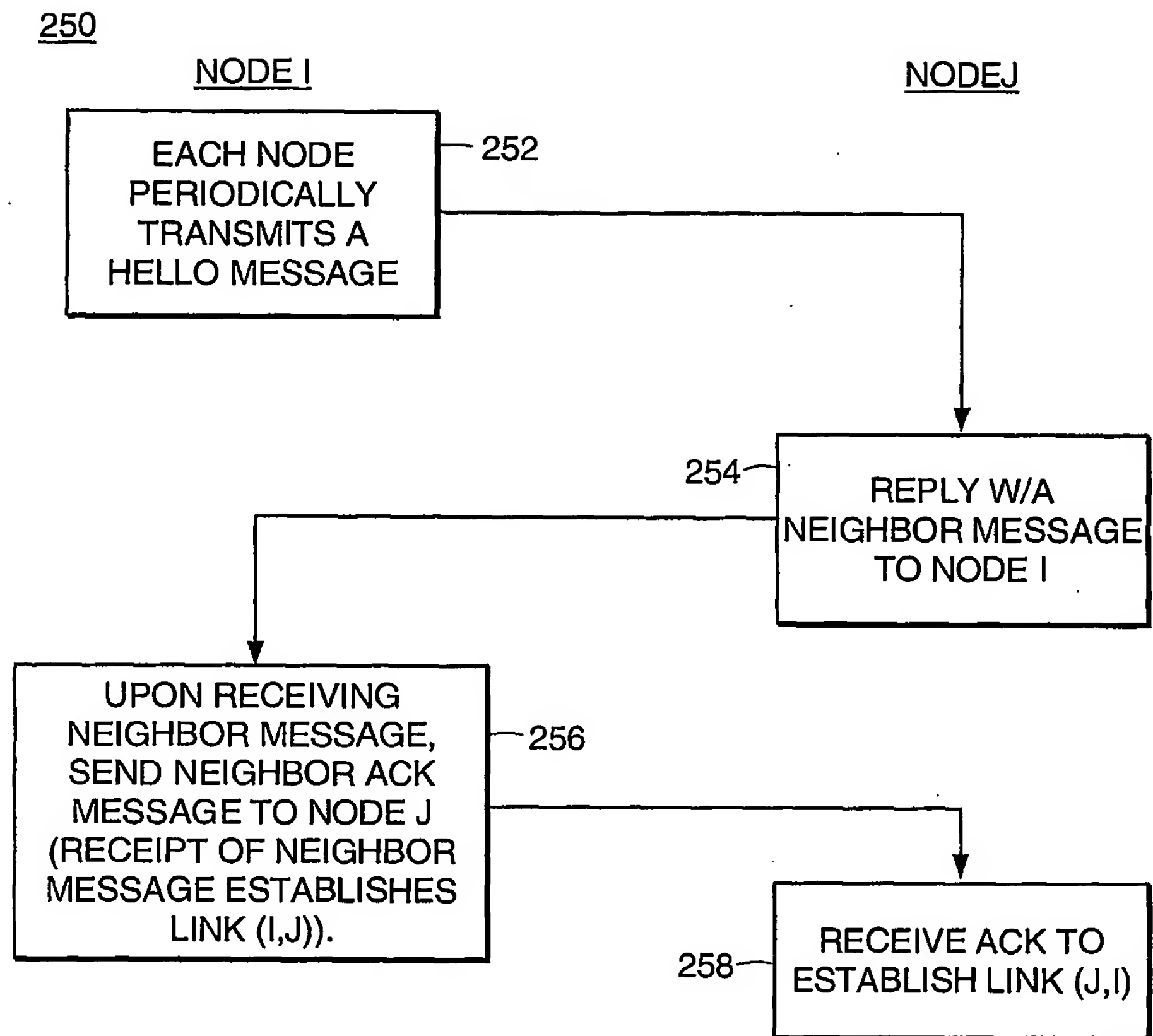


FIG. 12

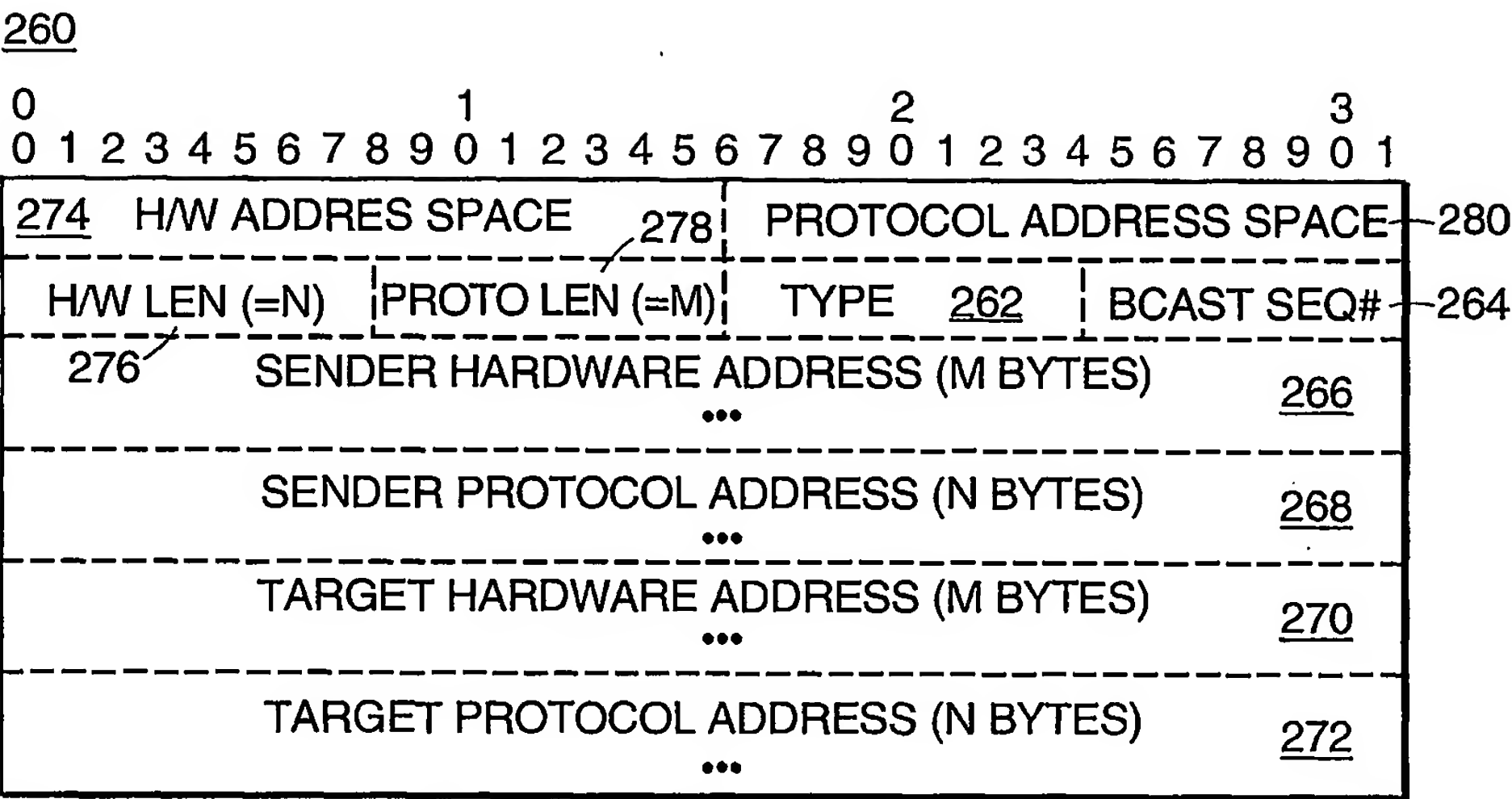


FIG. 13



15/20

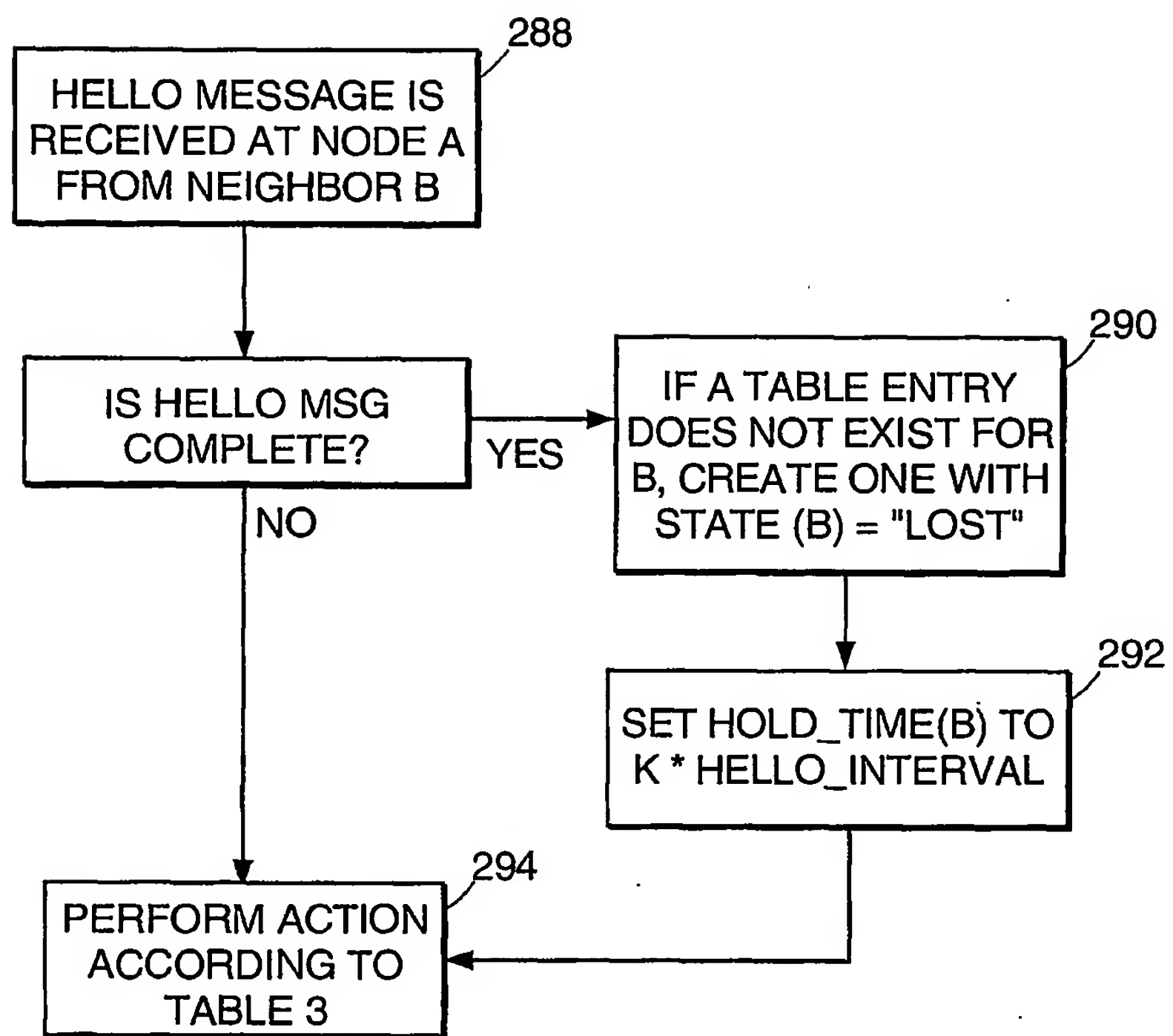


FIG. 14

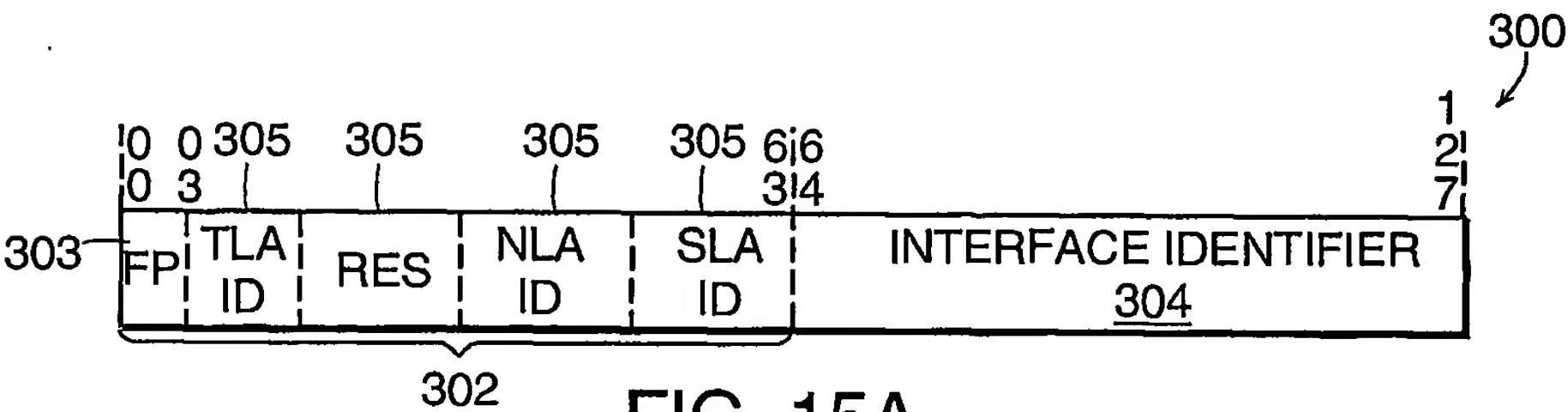


FIG. 15A  
IPv6-IPv4 COMPATIBILITY ADDRESS FORMAT

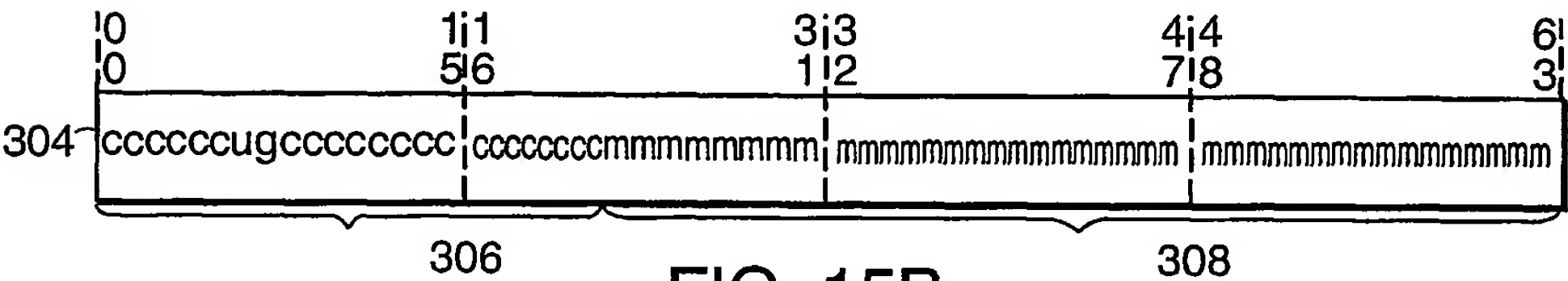


FIG. 15B  
INTERFACE IDENTIFIER FORMAT

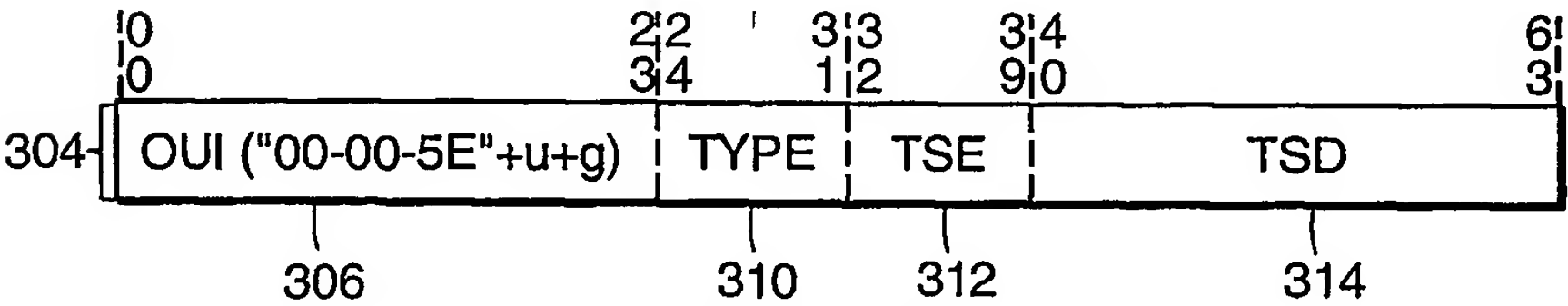


FIG. 15C  
INTERFACE IDENTIFIER FORMAT FOR IANA

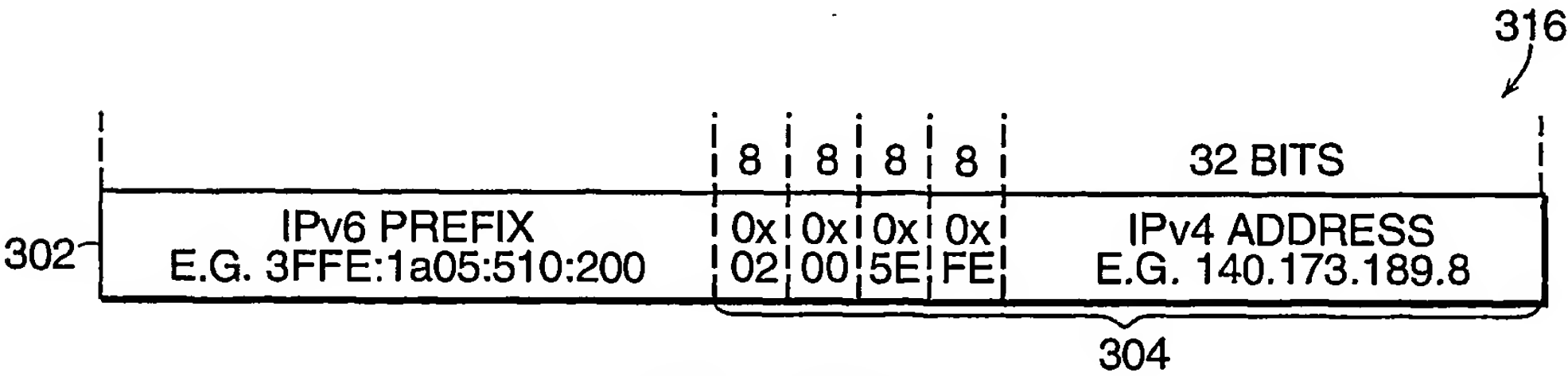


FIG. 15D  
IPv6-IPv4 COMPATIBILITY ADDRESS

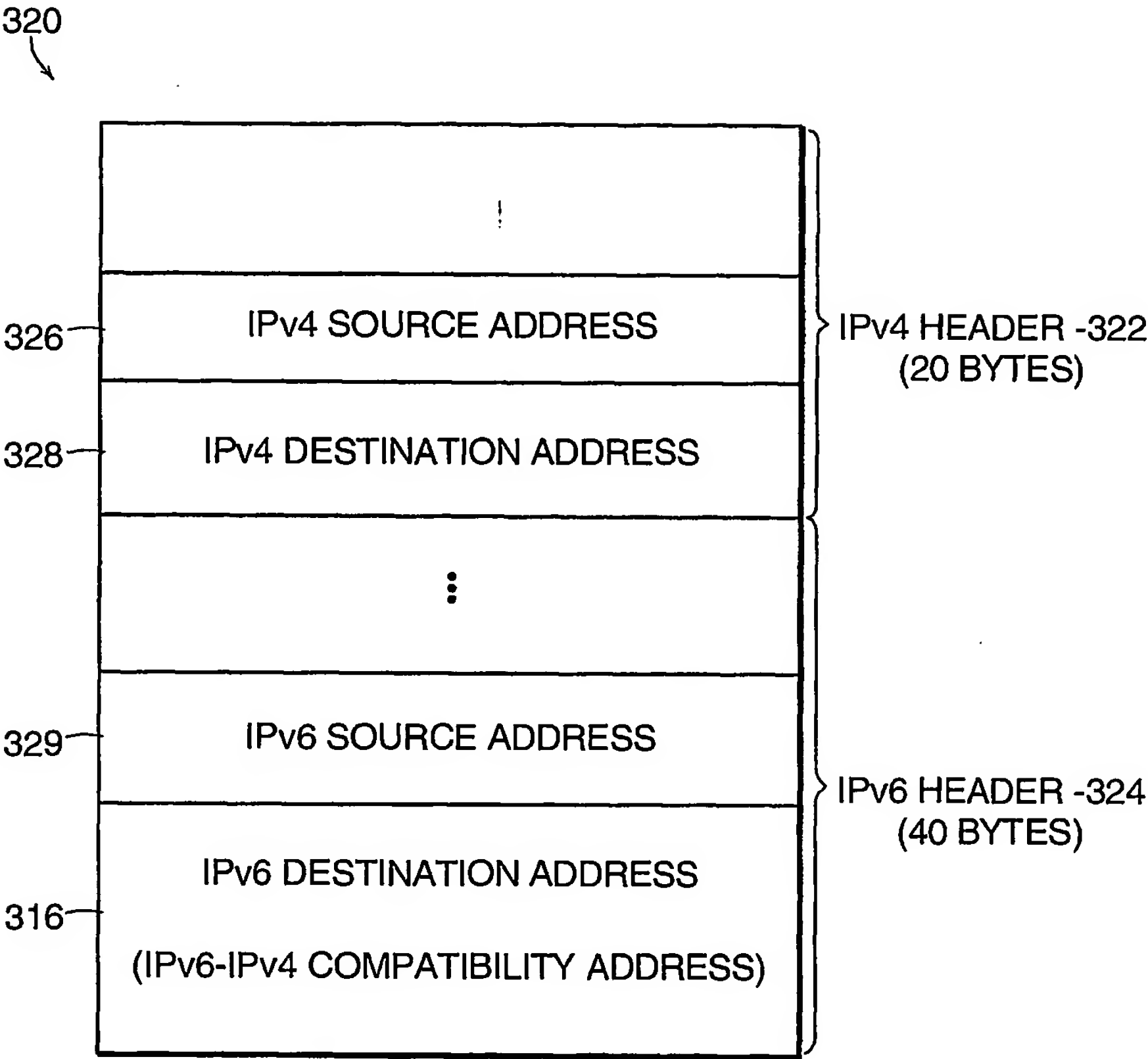


FIG. 15E

18/20

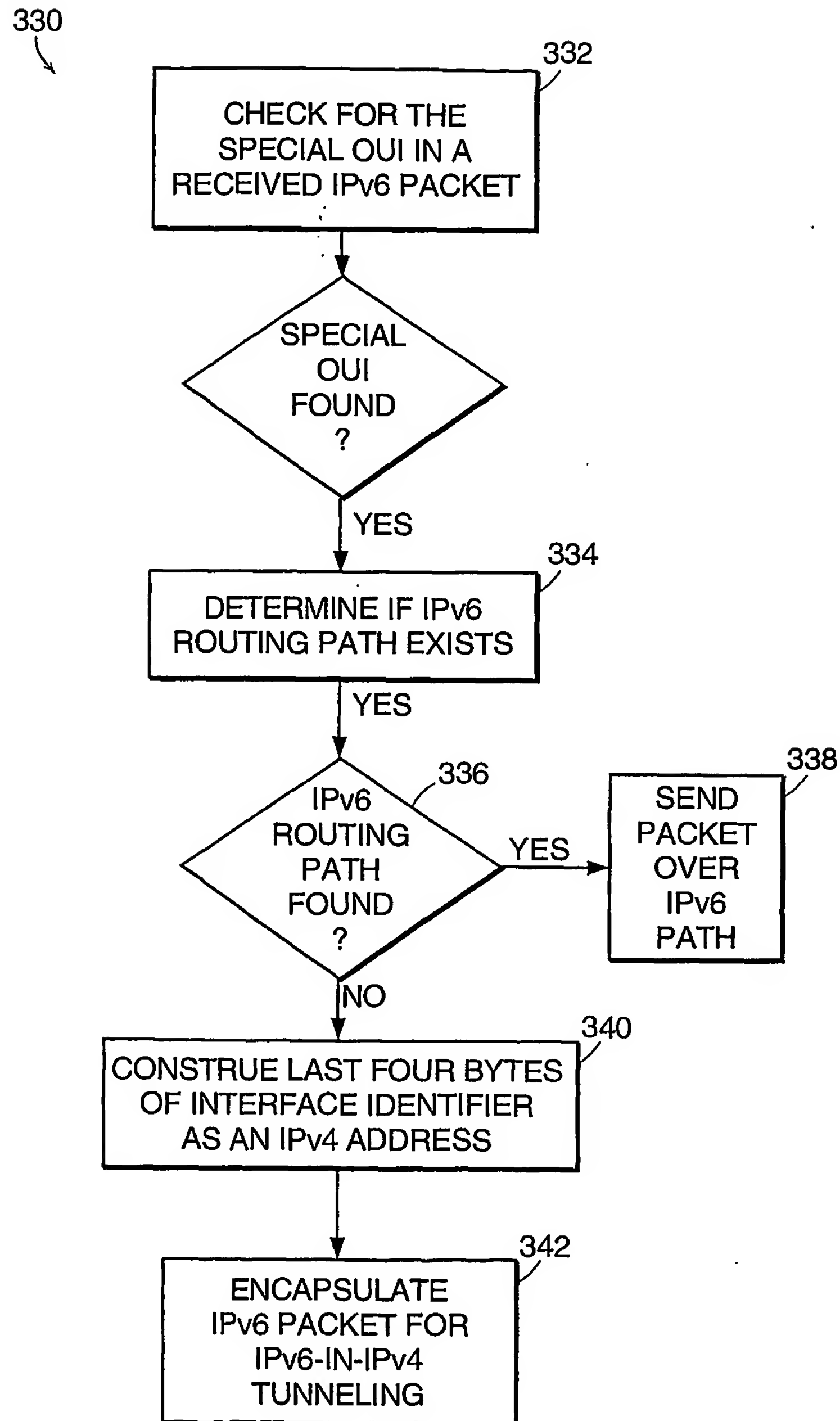


FIG. 16

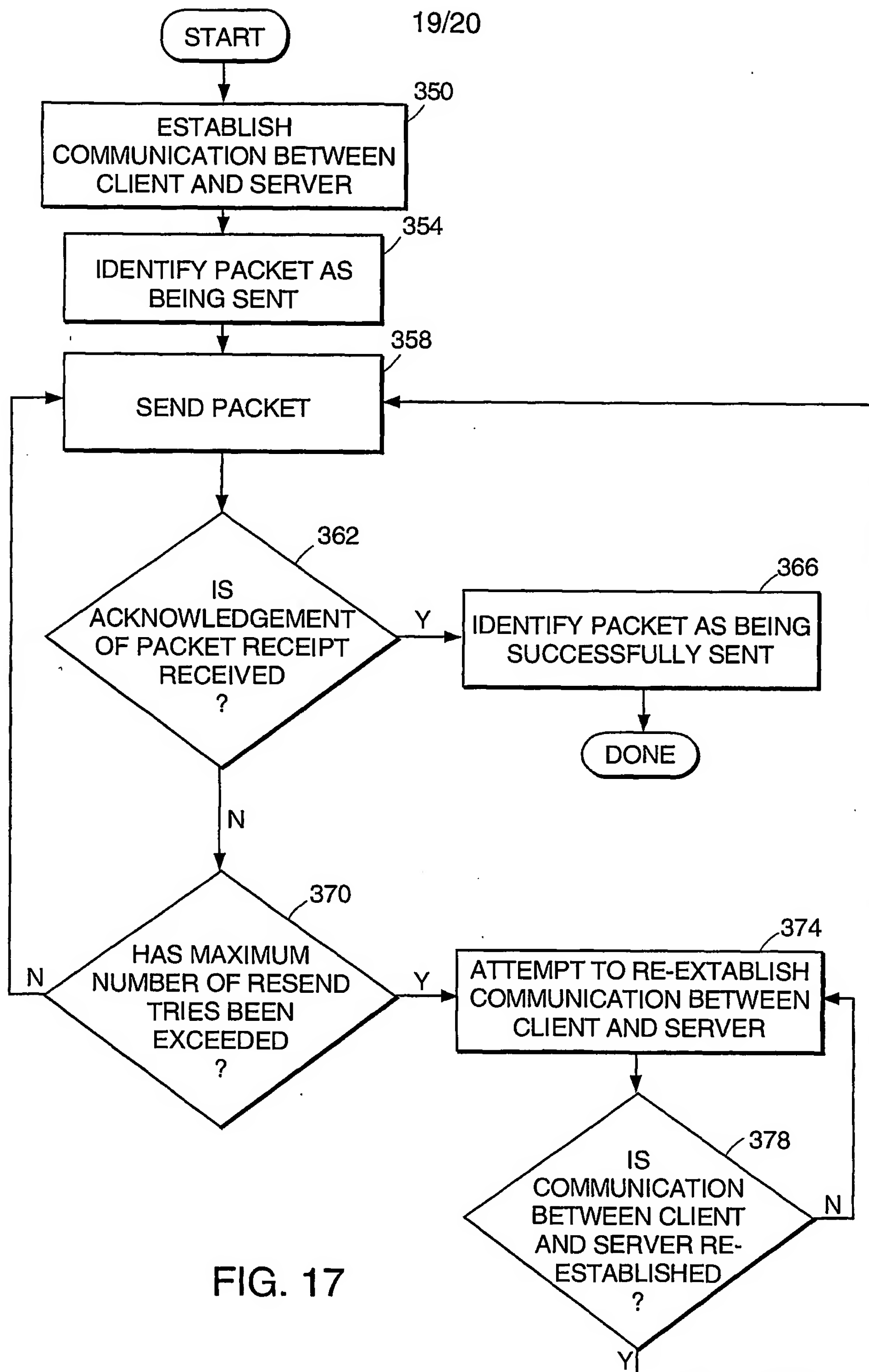


FIG. 17

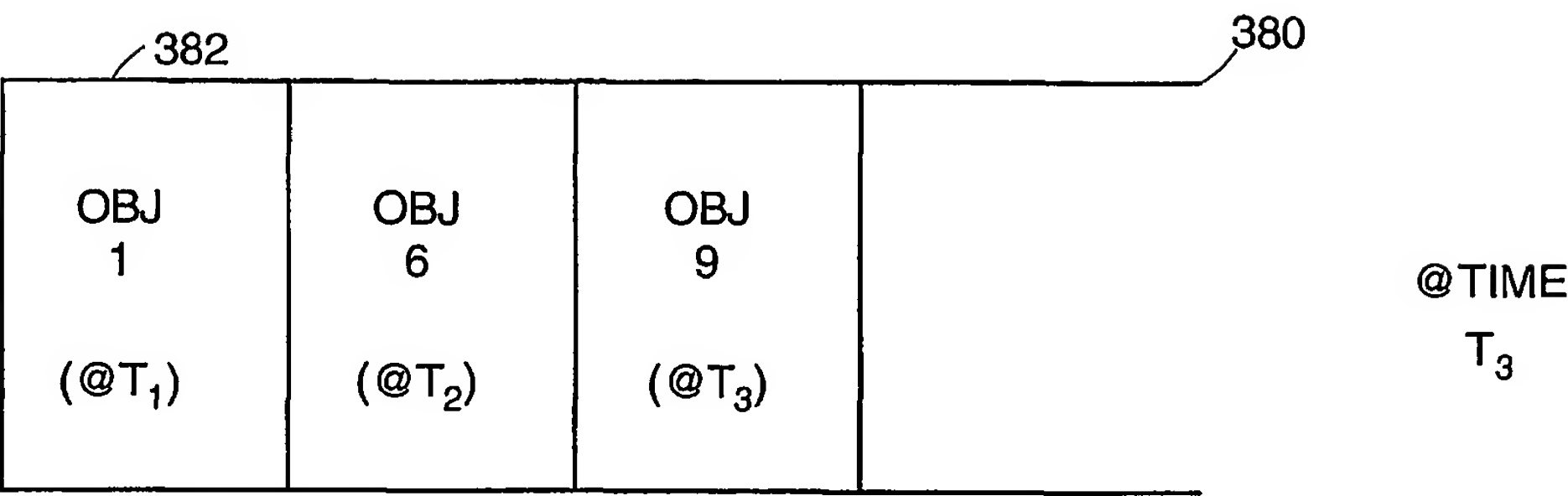


FIG. 18A

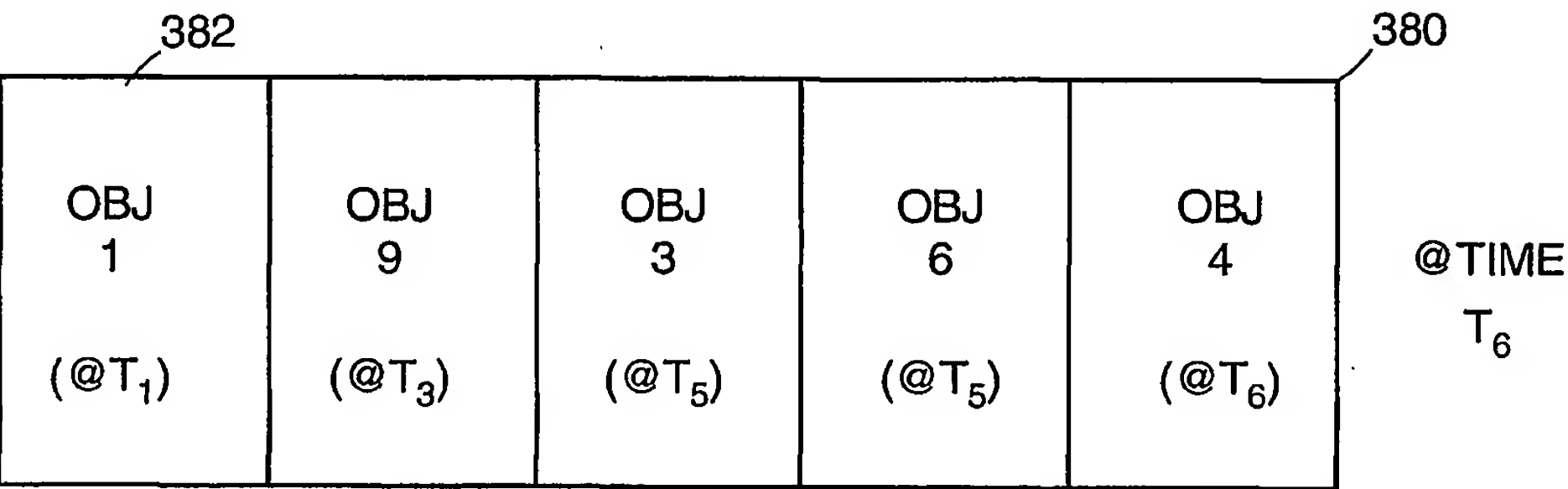


FIG. 18B